Developing a catalogue of errors and evaluating its impact on software development

Indrit Troshani
Edith Cowan University

Follow this and additional works at: https://ro.ecu.edu.au/theses

Part of the Software Engineering Commons

Recommended Citation

This Thesis is posted at Research Online. https://ro.ecu.edu.au/theses/1503
Developing A Catalogue Of Errors And Evaluating Its Impact On Software Development

Indrit Troshani
Edith Cowan University
You may print or download ONE copy of this document for the purpose of your own research or study.

The University does not authorize you to copy, communicate or otherwise make available electronically to any other person any copyright material contained on this site.

You are reminded of the following:

- Copyright owners are entitled to take legal action against persons who infringe their copyright.

- A reproduction of material that is protected by copyright may be a copyright infringement. Where the reproduction of such material is done without attribution of authorship, with false attribution of authorship or the authorship is treated in a derogatory manner, this may be a breach of the author’s moral rights contained in Part IX of the Copyright Act 1968 (Cth).

- Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
Developing a Catalogue of Errors and Evaluating its Impact on Software Development

By

Indrit Troshani

A Thesis Submitted in Partial Fulfilment of the Requirements for the Award of Doctor of Philosophy

At the Faculty of Computing, Health and Science, Edith Cowan University, Mount Lawley Campus

Date of Submission: 28/05/2003
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
Developing a Catalogue of Errors and Evaluating its Impact on Software Development

By

Indrit Troshani
Bachelor of Business Administration (Honours)
Master of Science (Computer-Based Information Systems)
Graduate Certificate of Education (Tertiary Teaching)

A Thesis Submitted in Partial Fulfilment of the Requirements for
the Award of
Doctor of Philosophy

At the Faculty of Computing, Health and Science, Edith Cowan University,
Mount Lawley Campus

Date of Submission: 28/05/2003
Abstract

The development of quality software is of paramount importance, yet this has been and continues to be an elusive goal for software engineers. Delivered software often fails due to errors that are injected during its development. Correcting these errors early in the development or preventing them altogether can, therefore, be considered as one way to improve software quality. In this thesis, the development of a Catalogue of Errors is described. Field studies with senior software engineering students are used to confirm that developers using the Catalogue of Errors commit fewer errors in their development artifacts. The impact of the Catalogue of Errors on productivity is also examined.
Declaration

I certify that this thesis does not, to the best of my knowledge and belief:

i) incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education;

ii) contain any material previously published or written by another person except where due reference is made in the text; or

iii) contain any defamatory material.

Signature: ____________________________

Date: 30/08/2004
Acknowledgements

I would like to acknowledge with gratitude the constructive advice and continuous encouragement and support of both my supervisors Dr Jim Millar and Dr Donald McDermid. I would particularly like to thank them for their patience and the hours spent during this project. Dr Millar provided enormous guidance and support since the early stages of this project including the definition of the research questions, project organisation and thesis writing. Dr McDermid's role was critical in the establishment and the presentation of the research approach and in the organisation and the writing of the thesis as an "argument", as he eloquently puts it.

I also would like to thank Dr Judith Rochecouste for her writing advice and her patience with the earlier drafts of chapters of thesis. I am grateful to Dr Danielle Brady for her advice on statistics and data analysis and presentation.

I am also thankful to the participants of the project presented in this thesis who helped by participating and by providing constructive feedback. I am thankful to Professor Galip Veliu and Professor Tony Watson who indirectly provided enormous support to me during the course of this project.

Finally, I would like to thank my partner Ruchi Permuvattana for her love and patience and for sharing the happy and the hard times during the period it has taken me to work on this thesis.
TABLE OF CONTENT

1. INTRODUCTION............................................................................................................ 9
   1.1 BACKGROUND ........................................................................................................... 9
   1.2 RESEARCH OBJECTIVE .......................................................................................... 10
   1.3 STRUCTURE OF THE THESIS.................................................................................. 10
   1.4 TERMINOLOGY ....................................................................................................... 12
      1.4.1 Definition of Error, Fault, and Failure .............................................................. 13
      1.4.2 Software Development Terms ......................................................................... 14
      1.4.3 Object-oriented Paradigm ............................................................................... 17
      1.4.4 Procedural Paradigm ...................................................................................... 18
   1.5 SUMMARY .............................................................................................................. 18

3. LITERATURE EXAMINATION ........................................................................................ 20
   2.1 OVERVIEW .............................................................................................................. 20
   2.2 ERROR DETECTION APPROACHES ....................................................................... 22
      2.2.1 Reviews and Inspections .................................................................................... 21
      2.2.1.1 Fagan’s Formal Inspections ......................................................................... 24
      2.2.2 Reading Techniques ......................................................................................... 27
         2.2.2.1 Ad-Hoc Reading ........................................................................................ 28
         2.2.2.2 Checklist-Based Reading ............................................................................ 28
         2.2.2.3 Scenario-Based Reading ............................................................................ 29
      2.2.3 Metrics ............................................................................................................. 31
         2.2.3.1 Using Metrics to Detect Errors in Requirements Artifacts ......................... 31
         2.2.3.2 Using Metrics to Detect Errors in Design & Code Artifacts ...................... 36
      2.2.4 Testing ............................................................................................................. 39
         2.2.4.1 Specification-versus Program-based Testing .............................................. 41
         2.2.4.2 Testing Different Levels of Abstraction ...................................................... 49
         2.2.4.3 Summary and Evaluation of Testing ............................................................ 62
      2.2.5 Summary and Evaluation of Error Detection Approaches ................................ 63
   2.3 ERROR PREVENTION APPROACHES ..................................................................... 65
      2.3.1 Preventing Errors in Requirements Artifacts .................................................... 63
         2.3.1.1 Goal Analysis ............................................................................................ 64
         2.3.1.2 Viewpoints Analysis ............................................................................... 67
         2.3.1.3 Use Case Analysis ................................................................................. 67
      2.3.2 Preventing Errors using Metrics ....................................................................... 68
         2.3.2.1 Formal Methods ....................................................................................... 68
      2.3.3 Cleanroom Software Development ................................................................... 69
      2.3.4 Preventing Errors using Metrics ....................................................................... 77
         2.3.4.1 The Personal Software Process (PSP) ........................................................ 78
         2.3.4.2 Haynes’s Phase-Containment Matrix ......................................................... 80
      2.3.3 Summary and Evaluation of Error Prevention Using Metrics ......................... 81
   2.4 CHAPTER SUMMARY AND DISCUSSION .............................................................. 82

3. RESEARCH QUESTIONS, SIGNIFICANCE, AND APPROACH ........................................... 87
   3.1 OVERVIEW .............................................................................................................. 87
   3.2 RESEARCH OBJECTIVE AND QUESTIONS ............................................................ 87
      3.2.1 Question One: The Error Framework ................................................................. 88
      3.2.2 Question Two: The Catalogue of Errors .............................................................. 89
      3.2.3 Question Three: The Impact of the Catalogue of Errors on Software Development ................................................................................................................................. 90
      3.2.4 Summary: Relationships between Questions ..................................................... 91
   3.3 MOTIVATION ........................................................................................................... 92
      3.3.1 Question One: Why develop an Error Framework? ........................................... 92
      3.3.2 Question Two: Why develop a Catalogue of Errors? ....................................... 93
      3.3.3 Question Three: Why investigate the impact of a Catalogue of Errors? ............ 94
11. APPENDIX A: CATALOGUE OF ERRORS ........................................................................ 231
12. APPENDIX B: INSTRUMENTATION ......................................................................... 475
13. APPENDIX C: DATA ................................................................................................. 537
14. APPENDIX D: SAMPLES OF ERRORS FOUND IN DEVELOPMENT ARTIFACTS... 555
1. Introduction

1.1 Background

In the recent years, the software development industry has witnessed an explosive growth. Every day, more organisations and businesses move from manual systems to innovative automated software-based systems in order to improve the efficiency, effectiveness, reliability, and the productivity of their operations. In this context, the development of high quality software is of paramount importance. While software quality is a highly desirable outcome of software development, for over three decades, software quality has been an elusive goal. This is because the software development industry has been greatly affected by what is commonly known as the software crisis (Conwell, Enright, & Stutzman, 2000; De Champeaux, 2002; Glass, 2002; Schulmeyer, 1990). Schulmeyer (1990) describes the software crisis by saying:

"Software development is in crisis. When delivered, computer software is often late, and it fails often because it contains defects. The answer to lateness and failures is quality." (p. xxiii)

De Champeaux (2002) suggests that the software crisis is still ongoing today:

"In spite of vigorous attempts and developments (OO, software development processes, development process metrics, DoEs, and UML, to name a few) since then [over 3 decades ago when the software crisis term was coined] the [software] crisis continues." (p. 102)

This indicates that additional research is needed to minimize the impact of the software crisis on the quality of developed software.

One of the most important aspects of software quality is related to errors that are committed and left undetected in the software during development (Fenton & Pfleeger, 1997; Kitchenham, 1996; Conte, Dunsmore, & Shen, 1986; Diaz & Sligo, 1997; Pfleeger, 1996; Wohlin, 1998). In the best case scenario, errors cause unwanted disruptions in the operations of organisations or businesses that rely on software-based systems, leading to customer dissatisfaction, low productivity and profit losses. In the worst case scenario, errors have life-threatening or disastrous consequences during the
operation of software-based safety or mission critical systems (e.g. the space shuttle or missile control systems, etc.) (Jezequel & Meyer, 1997; Sommerville, 1996). It follows that, in either case, it is imperative that either the generation of errors be prevented or that errors be detected and corrected during software development. However, in order for error prevention and/or detection to be successful, software developers need to know about errors. Knowledge of errors can play a crucial role in software development. The reason for this is that different types of errors betray a lot of information concerning their likely origin, cause, manifestation etc. Developers who have access to such knowledge would be more likely to make informed decisions about how to change their development practices to avoid errors (Freimut, 2001).

1.2 Research Objective

The objective of this thesis is to develop an error prevention approach, which is relatively simple to use, inexpensive, and suits the needs of individual developers. This can be done by identifying the way that error information can be used to prevent errors from being injected into software, or at least to help developers detect them as early as possible during, rather than at the end of software development. This research objective can be accomplished by investigating the following related areas:

i) What information about errors may be useful to developers in order to help them prevent and/or detect errors during software development? Why is such information important?

ii) Can information about different types of errors be catalogued in a systematic way? How can this catalogue of errors be improved to suit the needs of the developers who use it?

iii) Can a catalogue of errors help developers prevent and/or detect errors during software development? What other implications can the use of a catalogue of errors have on developers' performance (e.g. productivity)?

1.3 Structure of the Thesis

The objective of this section is to show how this thesis is structured. Following this introductory chapter, chapter two examines the related literature. This examination focuses on two broad approaches that can be used to target errors:

i) Error detection approaches, and
ii) Error prevention approaches.

Both of these approaches are divided further into subcategories. During the examination of error detection and prevention approaches, comparisons, contrasts, and critical evaluations are made.

In chapter three, the research objective and the three research questions that were addressed in this study and their relationships are clearly defined. The need for the research questions, the possible benefits which may result by addressing them, and the respective success factors are explained. In addition, in this chapter the research approach, which is a field study, is defined. The field study is described after relevant research methodologies are reviewed and compared.

In chapter four, issues relating to the design of the field study and its components (e.g. field experiments) are identified and addressed. These issues include:

i) Planning, which addresses participant selection, experimental, and instrumentation issues.

ii) Operation, which addresses participant preparation and data validation issues, and

iii) Data analysis methods, which summarise the ways in which the collected data will be analysed.

In chapter five, four aspects of the validity of the field experiments, which are the most important components of the field study, are addressed. They include conclusion validity, internal validity, construct validity, and external validity. A discussion evaluating the different types of validity and their impact on the field experiments concludes the chapter.

In chapter six, the first research question is addressed. This is concerned with the development of an error framework and its empirical evaluation via a questionnaire. The literature underlying the individual elements of the error framework is examined first. The motivation to unify the individual elements into a unique error framework is then presented. Following this, an error framework is proposed and contrasted with existing frameworks. The need for the empirical evaluation of the error framework is
then explained. The chapter concludes with the examination of empirical data which affirm the usefulness of the proposed error framework.

In chapter seven, the second research question is addressed. This question is concerned with the construction of a catalogue of errors, based on the error framework proposed in the previous chapter. Chapter seven outlines the process followed to develop the catalogue of errors and the differences between two versions of the catalogue of errors are identified. Finally, empirical data collected via a questionnaire, evaluating the two versions of the catalogue of errors are presented.

In chapter eight, the third research question is addressed. This research question deals with the empirical evaluation of the impact that the catalogue of errors has on the number of errors committed and corrected by software developers and their productivity. The empirical evaluation is carried out by using data collected from two field experiments.

Chapter nine discusses the outcomes and the implications of this research. The outcomes of each research question are revisited and the contributions of the thesis to this field of knowledge are highlighted. Limitations of the study and future research directions are also addressed.

1.4 Terminology

In this section some basic working definitions used in this thesis are outlined. As indicated in section 1.2, the research described in this thesis focuses on the study of errors, the development of a catalogue of errors and the examination of its impact on software development. In this context, two categories of terms are identified. First, error-related terms which include, errors, faults, failures, etc. will be defined in section 1.4.1. Second, software development-related terms which include development concepts and have been grouped in two subcategories, namely basic software development terms relating to the development phases and development paradigm terms.

As will be shown in chapter four, an object-oriented Java software development environment was used to assess the impact of the catalogue of errors on the quality of
delivered software and developer productivity. Consequently, it is important that some basic working object-oriented notions and terms be defined. In addition, from time to time in the discussion, comparisons between the object-oriented paradigm and the procedural alternative are made. This necessitates the explanation of some basic procedural paradigm notions as well. Software development-related terminology (including its two subcategories) will be addressed in sections 1.4.2 through to 1.4.4. The different categories of terms are summarised in figure 1.1.

1.4.1 Definition of Error, Fault, and Failure
This research focuses on the study of errors. In this thesis, the definition of errors and related notions is generally based on the taxonomy of definitions provided by Binder (2000) and Jørgensen (2002), who rely on the standard definitions developed by the Institute of Electronic and Electrical Engineers (IEEE) Computer Society (Binder, 2000; Chillarege, 1996; Jørgensen, 2002).

An error represents a human action that may lead to problems in the correct behaviour of software. A synonym of an error is a mistake. A fault is what results from an error. The terms defect and/or bug are common synonyms. More specifically, a fault is the mode of expression of an error. The mode of expression of an error can include narrative text in a textual software artifact, a dataflow diagram, source code, etc. Jørgensen (2002) illustrates an example of a fault:

"When a designer makes an error of omission, the resulting fault is that something is missing that should be present in the representation." (p. 3)
A failure occurs when the fault executes. Specifically, a failure is an observable deviation in the behaviour of software from the required capability. Examples include missing or incorrect output, unacceptable performance in time or space, or abnormal termination. The notions that were explained above have been summarised in Figure 1.2.

![Figure 1.2 - Errors, Faults, and Failures](image)

It should be emphasised that sometimes the above terms are used interchangeably (Mays, Jones, Holloway, & Studinski, 1990), inconsistently (Jorgensen, 2002), or even ambiguously (Binder, 2000). For example, the terms error and defect are used interchangeably in Gilb & Graham, (1996) and Mays et al., (1990); the term defect is used to refer to both faults or failures in Freimut, (2001); the term defect is used to refer to errors, faults, and failures (Lanubile, Shull, & Basili, 1998). Nevertheless, the categories defined above are generally found useful (Binder, 2000).

In this thesis, the definitions of error-related terms will adhere to the definitions that are shown in figure 1.2. Deviations in the use of the terms of figure 1.2 will be only made when it is necessary to be consistent with the terminology used by the original authors. Such cases will be duly acknowledged.

1.4.2 Software Development Terms

Software is developed by progressively refining a set of requirements into a working system (Pressman, 1997; Sommerville, 1996). Typically, this refinement occurs during designated development phases. Typical development phases include requirements
specification, design, code, and testing (Landis et al., 1990; Landis et al., 1992; Pressman, 1997; Sommerville, 1996).

During the requirements specification phase or simply the requirements phase, the needs of the users (or clients) are formalised into clear, unambiguous, correct, consistent, and complete statements detailing the services expected to be provided by the software and its constraints. The requirements are documented in the requirements artifact.

The design phase consists of two components, namely, high-level design and low-level design. High-level design specifies an overall model representing the structure of software. This model is also known as software architecture and is composed of individual elements which are also referred to as subsystems. Software architecture shows how individual subsystems interact with each other. Individual subsystems can be traced to the individual requirements specified during the requirements phase and are specialised to carry out specific services (also known as functions, operations, and behaviours). In order for such services to be successfully delivered to the final software user, a set of steps must be detailed. The steps represent individual instructions which provide a required service when carried out in the specified order. This collection of steps is also known as an algorithm. Algorithm definition is carried out during low level design. Each service must be easily traceable to a subsystem in the high-level design. The information that is produced from high-level and low-level design is documented in a design artifact.

During the coding phase the algorithms specified in low-level design are translated into executable programs and integrated into subsystems, as dictated by the high-level design. The collection of the executable programs produced during the coding phase represents the code artifact. Finally, the compliance of the individual services, the subsystems and the entire software to the original requirements (specified during the requirements phase) is validated during the testing phase. Software validation-related information is included in the testing artifact.
At the completion of each phase, an artifact is delivered which is further refined in the subsequent phase. For example, at the completion of the requirements phase a requirements artifact is produced. The requirements artifact is used as input to the design phase, at the completion of which a design artifact is generated. This process continues until a working software application is completed (figure 1.3).

In this thesis, the term phase will be used to refer to a part of the development where the development of an artifact occurs, whereas the term development artifact will be used to represent the deliverable resulting from work carried out during any development phase. Where specific development artifacts are dealt with, the term development artifact will be prefixed with an identifier, indicating the phase during which the artifact is produced. For example, requirements development artifacts or simply requirements artifacts are produced during the requirements phase, design artifacts are produced during the design phase, and so on.

The literature suggests that there are slight variations in the way development phases and artifacts are named and defined. In this thesis, the definitions provided earlier in this section will be adhered to. The relationships between the development phases and the respective artifacts have been summarised in figure 1.3.

![Figure 1.3 - Development phases and artifacts](image-url)
1.4.3 Object-oriented Paradigm

Binder (2000) emphasizes the necessity to define basic object-oriented terminology in that the: "Ambiguous usage, synonyms, and homonyms are rampant in object-oriented development, so definitions of basic terms ... are in order." (p. 18). Consequently, the following object-oriented paradigm working definitions are warranted.

The basic lexical unit of an object-oriented program is a class. A class defines instance variables to allow data storage and implements methods to manipulate the data stored in the instance variables. An instance variable is an attribute of a class that can be implemented using primitive data types, such as integer, byte, character, etc., or user defined data types (Binder, 2000). A method contains a "lexically contiguous unit of statements" (Binder, 2000, p. 18) that can be executed in response to a message. A message is a mechanism that allows a method to be invoked by associating it with a specific object. An object is a run-time instance of a class and the class is therefore said to be the template from which an object is instantiated. An object contains specific values for each instance variable of the class that it is instantiated from. Also, an object has access to the methods defined in the class and it can use them to manipulate values stored in the instance variables.

Among others, the object-oriented paradigm provides full support for inheritance and polymorphism (Binder, 2000). Inheritance constitutes the act of deriving a new class from an existing one (Lewis & Loftus, 2001). The new class is called a subclass. The terms, child class and derived class are synonyms with the term subclass. The existing class that is used to derive the new one is called the superclass. The terms, parent class and base class are synonyms of the term superclass. Together, a superclass and a subclass form an inheritance hierarchy. Technically, there is no limit to the number of subclasses that can inherit from the same superclass (i.e. the breadth of the inheritance hierarchy). Also, a class can be both a subclass and a superclass. For example, if class C inherits from class B, which inherits from class A, then class B is a subclass of A and a superclass of C. In this case, the inheritance hierarchy of classes A, B, and C contains three levels. Technically, there is no limit to the number of levels that an inheritance hierarchy can have (i.e. depth of inheritance hierarchy).
A subclass is said to inherit the instance variables and the methods of its superclass, which means that the subclass can access the instance variables and use the methods of its superclass. In addition, a subclass can specify its own new instance variables and methods. A subclass can also, override (or redefine) the superclass definition of an inherited method by retaining the method name and redefining the statements inside the method. This suggests that an implication of inheritance is that the same method (name) can have many forms (because the statements inside the method can be modified or redefined). This is known as polymorphism.

Programming languages that provide full support for the notions of classes and objects are called object-based programming languages. Object-based programming languages that provide support for the notions of inheritance and polymorphism are called object-oriented programming languages (Binder, 2000; Deitel, Deitel, & Nieto, 1999b; Louden, 2003).

1.4.4 Procedural Paradigm
As the name suggests, a program written in the procedural paradigm consists of a collection of procedures. A procedure is a logical sequence of statements or instructions, which when executed, deliver a unique service (Brookshear, 1997; Louden, 2003). A procedure is the basic lexical unit of a procedural program. A typical procedural program can have many specialised procedures which can have access to data and which manipulate data stored in variables common to all procedures. A typical procedural program also contains a main procedure whose objective is to control how other specialised procedures are executed.

1.5 Summary
This chapter has provided a background to the study, the research objective, including the general areas of investigation. The structure of the thesis has been presented, followed by a set of working definitions. The structure of the thesis is summarised and presented as a flow chart in figure 1.4.
Figure 1.4 – Thesis Flow Chart
2. Literature Examination

2.1 Overview
The objective of this chapter is to identify the various methodologies and techniques that have been used to target errors in software development. The examination of the literature has revealed that methodologies and techniques targeting a development artifact in particular or all development artifacts in general, adopt two different approaches towards errors (figure 2.1). The first is the detective approach, where an artifact is first developed and then errors that might have been injected during its development are detected using a set of predefined steps or criteria. The second is the preventive approach, where the development artifact is developed at the same time as being examined for errors. The detective approach is regarded as destructive in nature, because it attempts to 'break' the artifact after development to see whether it complies with its initial development requirements. The preventive approach is regarded as constructive because it aims to construct error-free artifacts (Coward, 1988).

![Figure 2.1 - Error Detection and Error Prevention](image)

Error detection approaches can be categorised into two further broad categories. First, non-execution-based error detection approaches examine development artifacts statically with the aim of locating injected errors. Second, execution-based error detection approaches examine development artifacts dynamically with the aim of locating injected errors. Typically, non-execution-based approaches are applied to both executable (e.g. code artifacts) and non-executable (e.g. requirements, design artifacts, etc.) software development artifacts, whereas the execution-based approaches are applied to only executable artifacts. The examination of the literature shows that there
exists a rich plethora of error detection approaches, both execution- and non-execution-based. In order to facilitate the presentation, the different error detection approaches have been categorised further, as shown in figure 2.2.

![Figure 2.2 - Types of Error Detection Approaches](image)

Reviews, inspections and reading techniques include systematic approaches to review development artifacts to identify errors. Metrics are concerned with measurements made to a development artifact with the objective of locating error-prone sections of the artifact. Testing offers techniques that help generate data, which can be used to run executable artifacts (i.e. code artifacts). The principal property of such data is that it helps expose the presence of errors in code artifacts.

In general, error prevention has focused on four main categories, which have been summarised in figure 2.3.

![Figure 2.3 - Summary of Error Prevention Approaches](image)
The first category focuses on error prevention approaches addressing requirements artifacts. The second category includes approaches which use error cause analysis in order to prevent errors. The Cleanroom software development uses a collection of techniques to prevent errors and constitutes the third category. Finally, errors can also be prevented by analysing metrics data, and the approaches that deal with this type of error prevention are classified under the fourth category.

The overwhelming conclusion from the literature is that the amount of work that has been carried out on error detection is significantly greater than that on error prevention. Work on error prevention, however, appears to have received more attention in the last decade. This is because error prevention is cheaper than error detection. Nevertheless, error detection and error prevention are closely related, because error detection contributes a great deal to our general knowledge about errors which is important for error prevention to be effective.

In section 2.2, the various error detection approaches (see figure 2.2) are discussed. The error prevention approaches (see figure 2.3) are discussed in section 2.3. Throughout the chapter the terms methodology or technique rather than the generic term approach will be used when specific methodologies and techniques are discussed. This is done in order to be consistent with the terminology used by the original authors. In section 2.4, the chapter is summarised and conclusions drawn.

2.2 Error Detection Approaches

The objective of this section is to survey the various approaches to error detection. As figure 2.2 shows there are four categories of error detection approaches. Each of these categories is discussed in the sections below.
2.2.1 Reviews and Inspections

Until the early 70s, it was believed that errors in computer programs could only be detected through program execution. This attitude prevailed until 1971, when Weinberg published his landmark book entitled “The Psychology of Computer Programming” (Weinberg, 1998). Weinberg uses the term “cognitive dissonance” to refer to the human tendency to self-justify actions. This means that a programmer will fail to detect any errors in the artifact that he/she has authored, no matter how blatant they are. This is because a programmer cannot see his/her own errors. However, a second programmer, who reads/reviews the same artifact, will pick the errors almost immediately. The notion of cognitive dissonance constitutes one of the most important aspects of software reviews.

The term software review suggests the examination of a development artifact. Weinberg’s notion of cognitive dissonance suggests that the review of a development artifact must be carried out by a developer other than its author. Since the publication of Weinberg’s work many types of software reviews, including technical reviews, software inspections, walkthroughs etc., have emerged in the literature. While these types of reviews are fundamentally similar, they are different with regard to the technical details of their practical implementation. A detailed characterisation of the most common types of software reviews is outside the scope of this chapter. Such details, however, can be found in Humphrey, (1989a) and Mazza et al., (1994).

The most popular software review technique is called software inspection. The software inspection method requires the visual inspection of a development artifact by a team of developers and the formal evaluation of the various work items included in the artifact through formal meetings (Chaar, Halliday, Bhandari, & Chillarege, 1993). The principal objective of software inspections is to detect errors as early as possible during software development (Chaar et al., 1993; Humphrey, 1989a; Wheeler, Bryczynski, & Meeson, 1996). In general, empirical research has shown that software inspections can identify up to 80% of all errors in the early phases of software development (Weller, 1993).

There are a few variations of software inspection, which are generally determined by factors such as, 1) the size of the team conducting the inspection, 2) number of meeting sessions where team members discuss the identified errors, 3) the method used to detect errors, and 4) post-error detection information collection feedback (e.g. error root cause
Due to space limitations, this section will only review Fagan's Formal Inspection method which appears to be the most popular in the industry. A detailed account of the different types of inspections is provided in Knight & Myers, (1993) and Porter, Sly, & Votta, (1996). The reason why only Fagan's Formal Inspection method is used is not only because this method is an error detection approach, but also because some of its variations introduce concepts (e.g. error abstraction) which have the potential to be used in error prevention.

2.2.1.1 Fagan's Formal Inspections

Fagan's formal inspections method was originally developed and used to manually review design and code artifacts at IBM. In Fagan's words, his inspection method was a "... formal, efficient, and economical method of finding errors in design and code [artifacts]" (Fagan, 1976). Since its introduction, Fagan's formal inspection method has been widely accepted and adopted in various organisations. Fagan's formal inspection method has also been adopted by IEEE as a standard (IEEE STD 1028-1988) (Wheeler et al., 1996).

Fagan's inspection method can be summarised to consist of three main steps. As a first step, the development artifact under consideration is inspected by individual reviewers with the purpose of identifying as many errors as possible. Henceforth, this is referred to as the error identification step. As a second step, the individual reviewers in a formal group setting are assigned well-defined roles and tasks and discuss the results of the error identification step according to a formalised agenda. Henceforth, this step is referred to as the group meeting step. Subsequently, during the third step, the identified errors are corrected by the author of the artifact. Henceforth, this is referred to as the error correction step. There is widespread agreement in the literature that formal inspections help detect a large number of errors. In fact, Fagan and others have shown that formal inspection can detect between 50 to 90 percent of the errors in a software artifact (Fagan, 1986; Gilb & Graham, 1996). The technical details of how formal inspections can be implemented can be found in Fagan, (1976); Gilb & Graham, (1996); Pfleeger, (1991); Schach, (1997, 1999) and Van Genuchten, Cornelissen, & Van Dijk, (1998).
Kelly et al. (1992) conducted a study of the density of errors found during the use of their customised variation of Fagan's formal inspection method at the Jet Propulsion Laboratory (JPL) at California Institute of Technology. The results of the research show a higher density of errors during the inspection of requirements artifacts. As progress was made, however, to the inspection of design and code artifacts, it was found that the error density decreased exponentially, because once errors in the earlier development phases were fixed, there was no opportunity for them to migrate to later phases of development. The empirical data, therefore, supported the conclusion that increasing the inspection rate in the early phases of the development decreases the density of errors found in later phases of the development (Kelly, Sherif, & Hops, 1992).

Doolan (1992) concludes that Fagan's formal inspection method can be applied to more than the design and code artifacts for which it was originally designed by Fagan in 1976. Fagan's formal inspection method can also be successfully used to detect errors in a range of other software development artifacts, including, requirements artifacts, scope documents, user documentation, test plans and results, management documents etc. (Doolan, 1992).

Schneider et al. (1992) adapted Fagan's formal inspection method on user requirements documents with the objective of locating requirements errors. They propose that Fagan's formal inspection method can be replicated to review user requirements documents, in parallel using N independent teams coordinated by a single moderator (Martin & Tsai, 1990; Schneider, Martin, & Tsai, 1992). This is based on the hypothesis that separate teams will not significantly duplicate each other's efforts by detecting the same error more than once, which was hinted at by the results of a pilot study carried out and discussed in Martin & Tsai, (1990). The findings of this empirical exercise revealed that, while formal inspections have been reported to detect 60 percent of the errors in design and code artifacts (Boehm, 1987a), they do not appear to be as effective in detecting errors in requirements artifacts. Out of 9 teams in an experiment by Schneider et al.'s (1992) only 4 managed to locate as many as 40 percent of the errors, with all teams on average managing to catch only 35 percent of the errors. While this result does not seem to be very encouraging, Schneider et al. (1992) claim that not a single error was found twice by the 9 teams, implying that using 9 teams did not duplicate the error identification effort. As a matter of fact, about 78 percent of the
errors were found by at least one of the nine teams involved. In addition, about 23 percent of all errors went undetected by all teams, which confirms that error detection is holistic in nature and must, therefore, be carried out throughout all phases of software development, starting from and including the requirements phase.

While the parallel replication is claimed not to affect the time required to complete the inspections, it cannot be denied that it will increase the human resources involved during the formal inspection activity, resulting in higher costs. In addition, there is no indication in Schneider et al. (1992) whether systematic measurements had been made to show that the additional cost incurred by using extra teams is actually offset by the savings resulting from the detection of the additional requirements errors. Such measurements are crucial, as they would have a bearing on the economic feasibility of the methodology.

In 1998, Lanubile, Shull, & Basili enhanced Fagan's original formal inspection technique to detect errors in requirements artifacts by including an additional step. This was inserted between the existing error identification and correction steps, and appears to replace the group meeting step (though this is not clearly shown in their publication), and entails the abstraction of the identified errors. Error abstraction requires the classification of the identified errors under abstract themes, free from domain specific details, in order to enhance the learning of errors by developers (Lanubile et al., 1998). For example, Lanubile et al. (1998) make reference to the requirements artifact of a Parking Garage Control System. For instance, if the requirement to display the number of vacant parking lots available were missing, then clearly the requirements artifact would contain an error. This error, which is associated with domain specific details, can be abstracted. The abstracted version could be called an omission error and be generically defined to represent a missing requirement.

Upon completion of the error abstraction step, the author of the requirements artifact is expected to receive the abstracted list of errors and use it to correct the requirements artifact. The inspection steps proposed by Lanubile et al. (1998) are repeated for each system requirements artifact requiring inspection. Lanubile et al. (1998) validated their variation of the formal inspection technique and reported positive results at a reasonable cost. However, despite these positive results, two drawbacks can be identified. First,
there are no clear guidelines on what would constitute a sound basis for error abstraction and how much information about an error can be abstracted. This may constitute a future research direction as error abstraction can help build up knowledge about errors which can be used later to prevent them. Second, the procedure described by Lanubile et al. (1998) suggests that error abstraction promotes the duplication of effort in the error abstraction documentation. This is because an abstract error documented for one requirements artifact will be identical to an abstract error documented for another requirements artifact.

In general, it is clear that significant research has been conducted on Fagan’s formal inspection method. While its different steps complement each other, the error identification step is the key step determining the effectiveness of the formal inspection to detect errors (Ciolkowski, 1999). Recent research has shown that the group meeting step does not increase the number of detected errors (Johnson & Tjahjono, 1998; Lanubile et al., 1998; Shull, 1998; Votta, 1993). Consequently, a meeting will not only fail to identify new errors but it may also not be always economically justifiable. It follows that, formal inspections can be as effective as the error identification step.

2.2.2 Reading Techniques
Significant research has been carried out concerning the techniques that an individual can use to review or read an artifact. Such techniques are commonly referred to as reading techniques. Individual reviewers apply reading techniques to read software artifacts and detect errors. Reading techniques can be applied to all artifacts, including requirements, design, and code artifacts to detect errors (Basili, Shull, & Lanubile, 1998; Basili, Shull, & Lanubile, 1999; Ciolkowski, 1999; Lanubile et al., 1998; Shull, Rus, & Basili, 2000; Shull, Travassos, Carver, & Basili, 1999; Shull, 1998; Travassos, Shull, & Carver, 1999c; Travassos, Shull, Carver, & Basili, 1999b; Travassos, Shull, Frederics, & Basili, 1999a). The literature shows many different reading techniques. The most popular ones include ad-hoc, checklist-based, and scenario-based reading techniques, which will be discussed in the following subsections.
2.2.2.1 Ad-Hoc Reading

With ad-hoc reading, reviewers inspect the whole software development artifact. They do not receive any guidance on what specifically to look for or how to proceed and error detection depends entirely on chance and/or the reviewer's experience, intuition, interest, and capability (Ciolkowski, 1999; Porter, Votta, & Basili, 1995). Reviewers do not require any training to learn how to use ad-hoc reading.

2.2.2.2 Checklist-Based Reading

With checklist-based reading, reviewers receive a list of questions which address various issues concerning the software development artifact under consideration (see table 2.1). Typically, the questions are dichotomous (e.g. yes/no) and are based on the prior knowledge of typical errors of the artifact.

<table>
<thead>
<tr>
<th>Table 2.1 — Excerpt of a Checklist (Porter et al., 1995)</th>
</tr>
</thead>
<tbody>
<tr>
<td>- Are the goals of the system defined?</td>
</tr>
<tr>
<td>- Are the requirements clear and unambiguous?</td>
</tr>
<tr>
<td>- Have the requirements been stated in terms of input/output and processing for each function?</td>
</tr>
</tbody>
</table>

It follows that the checklist questions can be customised for different artifacts and for different errors. Reliance on prior knowledge, however, may also constitute a problem, if reviewers do not pay attention to errors that have not been previously encountered and/or documented (Laitenberger, Emm, & Harbich, 1999a). Checklists provide only partial guidance to detect errors, because they guide the reviewers what to check for, without showing how to look for errors (Ciolkowski, 1999; Porter et al., 1995). Thus, reviewers who use the checklists may need some training to learn how to use them properly. There are other pitfalls that are attributed to checklist-based reading (Ciolkowski, 1999; Laitenberger et al., 1999a; Parnas & Weiss, 1985). In order to make checklist-based reading practical and not too time-consuming, checklist questions are designed to be general and, therefore, sometimes lack specific guidance. The lack of guidance also extends to not asking reviewers to document their analyses. Consequently, an individual reviewer's efforts may not be repeatable by others (Laitenberger et al., 1999a). The generality issue can be addressed by augmenting the
checklist with more specific questions; there is a downside, however, that is the checklist might become too lengthy, and therefore potentially tedious and time consuming to complete (Ciolkowski, 1999). Another issue concerning checklists is that they are standardised, and therefore multiple reviewers are likely to cover the same aspects and detect the same errors in a development artifact. Moreover, checklist-based reading requires reviewers to indiscriminately read all information in the artifact for possible errors. This might cause a reviewer to 'get swamped' with information that he or she may not understand or be interested in. Parnas & Weiss (1985) were among the first to identify this problem with checklists. Their active design reviews suggest that a development artifact should be reviewed by several experts in order to detect errors. Experts use specialised checklists with questions from their area of expertise. For example, an application domain expert focuses on application domain aspects of the artifact, and a user interface expert focuses on user interface aspects of the artifact (Ciolkowski, 1999). This means that multiple aspects of the artifact can be covered, which according to Parnas & Weiss (1985) provides an objective of reading techniques. While the active design review constitutes an improvement over checklists, it still suffers from one shortcoming which is inherent in checklist-based reading as well: it does not show how to look for errors. This is addressed in scenario-based reading, which is covered in the following section.

2.2.2.3 Scenario-Based Reading

Scenario-based reading suggests that a reviewer should read an artifact while following a set of guidelines, called a scenario (Ciolkowski, 1999). The guidelines not only help address what errors to look for, but also how to scrutinize an artifact in a systematic manner in order to locate errors (see table 2.2). Scenario-based reading techniques fulfill the active design review artifact multiple aspect coverage objective. Thus for scenario-based reading to be effective, several reviewers should apply several different scenarios in their reviews. Consequently, all or at least most aspects of the artifact and most errors are likely to be detected. Two main scenario-based reading techniques have been proposed and discussed in the literature: defect-based and perspective-based reading.
Table 2.2 – Excerpt of a Scenario (Porter et al., 1995)

<table>
<thead>
<tr>
<th>Incorrect Functionality Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. For each functional requirement identify all input/output data objects:</td>
</tr>
<tr>
<td>a) Are all values written to each output data object consistent with its intended function?</td>
</tr>
<tr>
<td>b) Identify at least one function that uses each output data object.</td>
</tr>
<tr>
<td>c) ...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Ambiguity or Missing Functionality Scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Identify the required precision, response time, etc. for each functional requirement</td>
</tr>
<tr>
<td>a) Are all required precisions indicated? ...</td>
</tr>
</tbody>
</table>

Defect-Based Reading

With defect-based reading, reviewers concentrate on scenarios whose guidelines focus on certain classes of errors. Specifically, the guidelines address issues of where and how to look for what errors. Currently, existing defect-based reading techniques are specialised to detect errors in requirements artifacts (V. Basili et al., 1998). Therefore, they target specific types of requirements errors such as, inconsistencies, ambiguities, incorrect or missing functionality¹, etc. The existing defect-based reading techniques can also be customised to deal with errors in other types of artifacts (e.g. design and code) (Ciolkowski, 1999).

Perspective-Based Reading

Perspective-based reading is based on the assumption that different stakeholders of a software development artifact value different aspects of that artifact in different ways (McCall, 1994; Shull et al., 2000). Consequently, perspective-based reading provides a set of individual reviews such that each review builds scenarios containing guidelines to cover the artifact from a different perspective, hence the name. The perspectives are determined by the stakeholders of the software artifact under review and may vary depending on the development artifacts and the organisation (Shull et al., 2000). The rationale behind perspective-based reading is based on the premise that the union of the perspectives provides an extensive coverage of the software development artifact. The implication of the extensive coverage is a better and more in-depth analysis of any

¹ There are examples of requirements errors. While the error name provides some idea about what the errors are about (e.g. ambiguity means that a certain requirement is ambiguous, therefore, not clearly stated), more precise definitions are provided in appendix A, sections 1.1 or 1.2 (Catalogue of Errors documentation).
errors in the artifact (Basili et al., 1996). While perspective-based reading has been shown to work with requirements artifacts, it can also be used with other types of development artifacts provided that the scenarios and the respective guidelines are tailored to the new perspectives (Ciolkowski, 1999). Basili et al. (1996) provide an example where a requirements artifact can be read from three perspectives, representing the respective stakeholders: the user of the system, the designer of the system, and the tester of the system. The scenarios in general, and the guidelines in particular, should consider the points of view or perspectives of the user, the designer and the tester of the system, if perspective-based reading is to be effective in detecting errors.

2.2.2.4 Empirical Evidence of the Effectiveness of Reading Techniques

The objective of this section is to present some empirical evidence about the error detection effectiveness of reading techniques. This evidence is important because it shows that reading techniques can be applied to all types of artifacts and consequently target all types of errors soon after the artifact has been developed.

Porter, Votta, & Basili (1995) report an experiment with graduate students as subjects, where the effectiveness of ad-hoc, checklist-based and defect-based reading techniques to detect errors in requirements artifacts was compared and contrasted. Admittedly, due to threats to external validity, the findings apply only to the experimental setting. The findings, however, have been widely cited in literature on rating the relative effectiveness of selected reading techniques. For example, Porter et al. (1995) found that the error detection rate of defect-based reading was 35 percent higher than ad-hoc or checklist-based reading techniques. It was also found that the performance of defect-based reading, which is a scenario-based reading technique (see section 2.2.2.3), depends on the way that the scenarios and the respective guidelines are designed. The results also showed that checklist-based reading was no more effective than the ad-hoc based reading. These results are corroborated by findings in a similar experiment with professional subjects (Porter & Votta, 1998).

In a different study, Ciolkowski (1999) reports the results of three experiments where students were employed to compare the error detection effectiveness of individuals and teams for requirements artifacts using checklist-based and perspective-based reading techniques. Unlike Porter et al. (1995) and Porter & Votta (1998), Ciolkowski (1999)
was unable to confirm any difference in the error detection effectiveness of individuals and teams for checklist and perspective-based reading techniques. This is attributed to the high degree of difficulty of the requirements artifact used in the experiment.

Ciolkowski's results, although inconclusive, have shown that requirements artifacts should be designed to be easily understandable by reviewers. In addition, the reviewers should be familiar with the application domain and be given a reasonable time frame to perform their reading. Reviewers also need appropriate training in the reading techniques (Ciolkowski, 1999).

Laitenberger & DeBaud (1997) report an experimental study where the effectiveness of perspective-based reading of code artifacts as a group activity is evaluated with professional developers from Bosch Telecom GmbH. The results indicate that the overlap of the detected errors reported by the different reviewers using different perspectives was relatively low, showing that there was effectively no duplication of effort with perspective-based reading (Laitenberger & DeBaud, 1997). The results also indicated that 'multi-individual' reviewer meetings identified only negligible numbers of errors (Laitenberger & DeBaud, 1997). Finally, the results also indicated that there were no significant differences with regard to the error detection rates of reviewers who had experience in the programming language and/or application domain of the code artifact and reviewers who did not (Laitenberger & DeBaud, 1997). This corroborates a similar finding which was reported in Basili et. al., (1996). They also emphasise the importance of tailoring the perspective-based scenario guidelines to the specific characteristics of the artifact under consideration and the organisation that has produced it.

In more recent studies, Laitenberger et al. (1999, 2000), compare and contrast the effectiveness of checklist- and perspective-based reading for design and code artifacts in experimental and quasi-experimental settings where software professionals were involved (Laitenberger, Atkinson, Schlich, & El-Emam, 2000b; Laitenberger et al., 1999a). The results indicated that perspective-based reading was more effective than checklist-based reading. Moreover, Laitenberger et. al. (1999; 2000) also found that the cost per error using perspective-based reading was significantly lower than checklist-based reading. Other empirical studies where similar findings have been reported can be found in Lanubile & Visaggio, (2000); Shull et al., (1999); Thelin &
2.2.2.5 Reading Techniques: Summary and Evaluation

In summary, this section compares and contrasts the reading techniques discussed above against five criteria, which have been defined below as:

i) Systematic: This criterion is concerned with whether the steps of the individual process to review an artifact for errors are definable or not.

ii) Focused: This criterion is concerned with whether the reviewers focus on the same or different aspects of the artifact.

iii) Controlled improvement: This criterion is concerned with whether reviewers can use feedback to identify and enhance one or more aspects of a reading technique.

iv) Customisable: This criterion is concerned with whether a reading technique can be tailored to specific artifacts (e.g. requirements, design, and code), paradigms (e.g. procedural, object-oriented) or organisational needs.

v) Training: This criterion is concerned with whether reviewers need training to successfully use a reading technique (Shull et al., 2000).

The evaluation of the reading techniques discussed above with respect to Shull et al. (2000) has been presented in table 2.3.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ad-hoc</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Checklist-based</td>
<td>Partially</td>
<td>No</td>
<td>Partially</td>
<td>Yes</td>
<td>Partially</td>
</tr>
<tr>
<td>Defect-based</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Perspective-based</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

As far as error detection effectiveness is concerned, it can be concluded that, in general, scenario-based reading is more effective and has a lower error detection cost than the other reading techniques (e.g. ad-hoc or checklist). Mainly, this is attributed to the fact that scenario-based reading techniques (i.e. defect-based, and perspective-based) tend to force reviewers to examine the artifact under consideration in more detail. While this may require the individual reviewer to put in more effort, it enhances the reviewer's understanding about the artifact under consideration and reduces the overall cost of
error detection. In addition, reading techniques in general and scenario-based reading techniques in particular appear to have a wide applicability to all of the main software development artifacts. When applied, however, they need to be carefully customised to suit the characteristics of the artifact under consideration, the application domain, the organisational needs etc.

2.2.3 Metrics
A metric is a quantitative measure of the degree to which a development artifact possesses a given attribute (IEEE, 1993). In this definition, a given attribute constitutes anything that a developer might be interested in knowing about a development artifact. Examples of metrics include the number of errors in an artifact, the complexity or length of the code, etc. When the value of a metric is measured for an artifact, developers can learn more about the artifact; they can rate it and compare it with other artifacts. There is widespread agreement in literature that some metrics, which are referred to as ‘amber’ or ‘warning’ metrics (Doake & Duncan, 1998), can be applied to software development artifacts with a great benefit: they help identify errors or parts of the artifact that are likely to contain errors. If the metrics are applied to artifacts that are developed in the early phases of software development (e.g. requirements specification, design) then their benefit can be even greater because errors will be identified early in the development. Early identification of errors is desirable because they can be eliminated with relative ease and low cost (Doake & Duncan, 1998; Duncan & Doake, 1998; Fenton & Pfleeger, 1997; Abreu & Carapuca, 1994).

The review of literature shows that several studies have been carried out in this area. Some have been empirically validated, while others have not. The surveyed studies can be grouped into two categories. First, those using metrics to detect errors in requirements artifacts. Second, those using metrics to detect errors in design and code artifacts. In the second category, studies about design and code have been deliberately grouped together because the literature suggests that metrics are used in the same way to detect errors in both design and code artifacts.
2.2.3.1 Using Metrics to Detect Errors in Requirements Artifacts

In 1993, Davis et al. argued that the presence or lack of errors in requirements artifacts can be signalled by measuring the values of a set of 24 attributes which they defined. These attributes include unambiguity, completeness, correctness, consistency, conciseness. According to Davis et al. (1993b), a given value of the unambiguity attribute indicates the lack or presence of ambiguity errors; a given value of completeness indicates the lack or presence of omission errors, and so on. The values of 18 of the 24 proposed attributes can be measured by computing 18 different formulae. Each formula requires data before its computation can be carried out. Davis et al. (1993b) suggest that the required data may be collected during reviews of the requirements artifacts for which the attributes are measured. No formulae have been proposed for 6 of the attributes, since the determination of the values these should be made subjectively. An example of how these metrics work can be provided by showing how the amount of ambiguity errors is measured. Ambiguity errors in requirements are measured as a percentage of the requirements that have been given identical interpretation by all reviewers. Such percentage values are expected to range from zero, where each requirement in the document would have multiple interpretations, to one, where each requirement would have a unique interpretation.

The inconsistency attribute constitutes another example. Consistency exists if and only if no one requirement stated in the artifact conflicts or contradicts any other requirement stated in the same artifact (Davis et al., 1993b). To measure consistency, the review of the requirements artifact must generate data on the count of unique requirements (n_u) and on the count of situations (n_s) where the software described in the requirements artifact is expected to assume different states as a result of the same input or interaction with its environment. Davis et al. (1993b) state that n_u count generation relies on the software (that is described in the requirements artifact) being depicted as a finite state machine (FSM). An FSM^2 is an analysis method whereby software requirements are mapped in terms of inputs, outputs, states, transitions, triggering events etc. When n_u and n_s are obtained, consistency can be computed by the following formula:

\[ \text{Consistency} = \frac{(n_u) - (n_s)}{(n_u)} \]

^1 FSMs are examined in more detail in Beizer, (1990, 1995).
The values for consistency can range from zero, which would indicate the presence of inconsistency errors in the requirements artifact, to one, which would indicate a lack of inconsistency errors in the requirements artifact.

The metrics proposed by Davis et al. (1993b) offer a rather long-winded approach towards error identification. While the values generated by the formulae may be helpful to judge the quality of a requirements artifact, they can only be obtained after the requirements artifact has been reviewed and using data that has been generated by the review. Therefore, any errors in the requirements artifact that are picked by its review will be reflected in the computed values of the formulae. In this context, it is not clear how much additional value the computation of the formulae provides to aid error detection. In addition, the computed values of the formulae do not contain any information about error location, which would be essential when errors are eventually corrected. Furthermore, it seems that for certain formulae to work (e.g. unambiguity computation formula), more than one reviewer is required. This has the potential to increase the cost of the reviews and error detection. Another drawback is that not all required data appears to be easily obtainable from simple reviews made to requirements artifacts. Certain formulae (e.g. inconsistency computation formula) suggest that requirements must be analysed by using specialised methods (e.g. FSMs), which sometimes cannot be used and sometimes can be too complex and time-consuming. It follows that complete reliance on certain methods of analysis would make some formulae useless, if software cannot be analysed by using such methods of analysis (e.g. inconsistency computation formula relies on FSMs). Finally, there is no indication in Davis et al. (1993b) that the proposed formulas have been empirically validated.

2.2.3.2 Using Metrics to Detect Errors in Design & Code Artifacts

Significant empirically validated work on the role of metrics to detect errors in design and code artifacts has been carried out by Basili et al. (1995; 1998) and Briand et al. (1997; 1999; 2000). These scholars and others have conducted several studies investigating the relationship between object-based and object-oriented design and code metrics, on the one hand, and errors in object-based and object-oriented design and code artifacts, on the other. A list of these studies including the types of metrics used, the conclusions made and their applicability, and the nature of the project where the validation data was collected from, has been summarised in table 2.4. In these studies,
the way the design/code classes are related to each other appears to influence the error-proneness of the design/code artifact.

It can be concluded from this table that, generally, there is an agreement that metrics can help identify errors or error-prone components. Coupling between classes\(^3\), in general, and the frequency of method invocations between classes, in particular, and the depth of inheritance hierarchies were found to be very strong indicators of error-prone classes. The surveyed studies were consistent in concluding that the principal benefit of using metrics is that errors or error-prone classes can be identified early in software development, resulting in development cost savings and allowing testing resources to be budgeted and allocated accordingly (Briand, Daly, Porter, & Wust, 1998a; Briand, Wust, Daly, & Porter, 2000; Fenton & Pfleeger, 1997). An exception to this general conclusion may include metrics derived from code artifacts. Their benefit may be of a lower magnitude as opposed to their pre-code (i.e. requirements, design metrics) counterparts due to the fact that code metrics are only obtained almost at the end of software development.

Liggesmeyer (1995) points out that the fact that an error-prone class is identified does not necessarily mean that the actual error or errors are revealed. Some may consider this inability of metrics to detect the actual error(s) as a disadvantage of metrics. However, the benefit of using metrics should be obvious from the cost perspective, when error detection resources can be allocated to error-prone classes, rather than to the entire software (Liggesmeyer, 1995).

---

\(^3\) Coupling between classes indicates the extent to which one class depends on another class. Depending on the degree of such dependence, coupling can be a design error. A more concise definition of coupling (and other design errors) and the situations when coupling can be an error is provided in appendix A, sections 1.1 or 1.2 (Catalogue of Errors documentation).
Table 2.4 – Summary of empirical studies evaluating the effectiveness of metrics to predict error and error prone components

<table>
<thead>
<tr>
<th>Source</th>
<th>Metrics used</th>
<th>Conclusions</th>
<th>Applicable to:</th>
<th>Validation context</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Basil, Briaud, &amp; Male, 1999)</td>
<td>Six Chidamber &amp; Kemerer (CK) Metrics (Chidamber &amp; Kemerer, 1994)</td>
<td>1) All but one (LCOM) of the CK metrics were successful in predicting error-prone classes. &lt;br&gt;2) Successful CK metrics were complementary to and independent of each other.</td>
<td>OO Design</td>
<td>University Project(s)</td>
</tr>
<tr>
<td>(Harrison, Sammarwayne, Dobie, &amp; Lewis, 1996)</td>
<td>Code metrics (e.g., no. of non-comment source lines, no. of domain specific, declared, defined, and library functions, depth of hierarchy)</td>
<td>1) All but the number of library functions and the depth of the hierarchy metrics significantly related with modification request made during testing and maintenance &lt;br&gt;2) Generally, the number of non-comment lines metric and the number of function declarations appear to be strongly related with quality of oo programs &lt;br&gt;3) Should be used after the CODING phase is complete</td>
<td>OO Code</td>
<td>Industry Project(s)</td>
</tr>
<tr>
<td>(Briaud, Devanbu, &amp; Male, 1997)</td>
<td>Coupling Metrics (considering type of relationship and the interaction between classes, etc.)</td>
<td>1) Coupling metrics were found to be significant predictors of fault prone classes. &lt;br&gt;2) Coupling metrics were found to be complementary to the CK metrics as predictors of fault prone classes.</td>
<td>OO Design</td>
<td>University Project(s)</td>
</tr>
<tr>
<td>(Sherif &amp; Santacrossa, 1998)</td>
<td>3 of the Chidamber and Kemerer (CK) Metrics (Chidamber &amp; Kemerer, 1994)</td>
<td>1) 3 CK metrics were used to compare the complexity of different projects. &lt;br&gt;2) Higher complexity classes were found to host more errors than lower complexity ones.</td>
<td>OO Design</td>
<td>Industry Project(s)</td>
</tr>
<tr>
<td>(Basil, Briaud, &amp; Maronce, 1998)</td>
<td>Object-based Coupling and Cohesion Metrics (similar to the ones covered in Briaud, 1997), excluding coupling based on inheritance relationships.</td>
<td>1) Found that using object-based coupling and cohesion metrics was helpful in predicting error-prone classes. &lt;br&gt;2) Do not guarantee that these types of metrics will have the same predictability for all application domains. Suggest that the predictability power of these metrics may depend on the application domain and environment.</td>
<td>OO Design</td>
<td>Industry Project(s)</td>
</tr>
<tr>
<td>(Yang, Kao, &amp; Chen, 1999)</td>
<td>5 Chidamber and Kemerer Metrics (CK) (Chidamber &amp; Kemerer, 1994)</td>
<td>1) Two out of the 5 CK metrics tried were found to be good indicators of erroneous classes and object-oriented errors.</td>
<td>OO Design</td>
<td>Industry Project(s)</td>
</tr>
<tr>
<td>(Briaud, Wust, Ivanovski, &amp; Lounis, 1999b)</td>
<td>Coupling and Cohesion Metrics proposed in: Briaud, Daly, &amp; Wust, (1998); Briaud, Daly, &amp; Wust, (1999a)</td>
<td>1) Coupling (especially focusing on the frequency of method invocations and the impact of change flow (e.g., a change to a class flows towards its subclasses)) was found to be a strong and stable indicator of errors. &lt;br&gt;2) Cohesion was not found to be a very good error-proneness indicator &lt;br&gt;3) Validity of error-prone indicators may be context-sensitive (e.g., application domain dependent)</td>
<td>OO Design</td>
<td>Industry Project(s)</td>
</tr>
<tr>
<td>(Briaud, Wust, Daly, &amp; Foster, 2000)</td>
<td>Coupling and Cohesion Metrics proposed in: Briaud et al., 1998; Briaud et al., 1999a</td>
<td>1) The number of dimensions captured by the metrics is lower than the number of metrics themselves, suggesting many metrics may be redundant. &lt;br&gt;2) Coupling (particularity, coupling induced by method invocations) and inheritance metrics (particularly, specialization changed and depth of inheritance hierarchy) are strongly related to the probability of error detection in classes. &lt;br&gt;3) Cohesion was not found to have an impact on error proneness.</td>
<td>OO Design</td>
<td>University Project(s)</td>
</tr>
</tbody>
</table>
A problem identified with some metrics, especially the ones proposed in Briand et al., (1998); Briand, Daly, & Wust, (1997) relates to the fact that some appear to be based on comparable ideas. The implication here is that different metrics predict identical errors or error-prone classes and are, therefore, redundant. In addition, many authors appear to agree that the error-proneness predicting ability of the metrics depends on the application domain and, therefore, developers should be cautious when generalising the reported findings, without replicating them in different application domains first (Briand et al., 1998a; Briand et al., 2000). Another problem that can be identified is that they can be paradigm-specific. For example, most of the metrics that are discussed in Briand et al., (1998); L. C. Briand et al., (1997) are only specific to the object-oriented design artifacts and cannot be readily applied to artifacts developed in different paradigms (e.g. procedural designs).

2.2.4 Testing

Humphrey (1989) defines testing as the execution of a program with the objective of finding errors. Seminal work on testing dates back to 1976, when Myers postulated a set of axioms for testing software in general and procedural software in particular. Myers’ axioms suggest that in order to expose the presence of errors in a program, the tester has to systematically generate test data and the respective output expected to result from the execution of the program under test with such test data. The test data and the expected output are collectively known as test cases. The test cases should provide for valid and invalid input conditions and are considered to be successful if they show the presence of errors, rather than their absence (Myers, 1976).

In 1989, Weyuker extended her previous work (1986) by abstracting and formalising a set of eleven testing axioms. Of the eleven axioms, seven are applicable to procedural software, whereas the remaining four are concerned with object-oriented software (Weyuker, 1986, 1989). Weyuker (1989) describes the difference between existing work and her own work thus:

"The philosophy behind this work is that software testing is more than just the selection of test data and the execution of software on that test set. We need to evaluate test data by using adequacy criteria and assess proposed criteria." (p. 668)
In other words, Weyuker's axioms describe criteria for the selection of the test cases that would adequately test a program to expose the presence of errors. Weyuker's axioms "... are, in some sense, negative axioms in that they expose inadequacy [of test cases], rather than guarantee [their] adequacy" (Perry & Kaiser, 1990, p. 14) (Perry & Kaiser, 1990).

The examination of literature shows that while comprehensive and substantial work has been done in the area of software testing, there exists a so-called negative reward structure of testing (Vick & Ramamoorthy, 1984). This means that the attitude of a dedicated software tester who attempts to break software in order to detect errors and report them often conflicts with the attitudes of the software developers and their managers who attempt to construct and deliver error-free software. Nevertheless, testing has been found to be critical to software development and efforts have been made to formalise a testing theory (Cherniavsky & Statman, 1988; Goodenough & Gerhart, 1975; Weyuker & Ostrand, 1980).

Weyuker & Ostrand (1980) capture the essence of the testing theory by stating that:

"The primary goals of a theory of testing are to provide a basis for practical program testing methodologies, and to establish ways of determining the effectiveness of tests in detecting program errors."

(p. 236)

In this context, however, Roper (1994) argues that:

"... while testing theory has made a contribution to testing research by exposing fundamental problems, it has not reached a stage whereby it may be used in the way suggested by Weyuker and Ostrand (1980)."

(p. 15)

This is because the testing theory constantly depends on a program (or part of it) being correct. And it is widely accepted that proving program correctness for a reasonable program can be an unreasonable proposition due to the need for exhaustive testing. This is commonly referred to as untractability of testing (Roper, 1994). Despite Roper's observation regarding the lack of a basis for practical software testing methods, a rich plethora of testing methodologies and techniques have been developed. These are discussed in the following sections.
2.2.4.1 Specification-versus Program-based Testing

Depending on the criteria used to generate test cases to expose errors, testing methods and techniques can be classified into two broad categories (Bashir & Goel, 1999; Beizer, 1990, 1995; Binder, 2000; McGregor & Sykes, 1992; Myers, 1979; Pressman, 1997; Roper, 1994). Specification-based testing (also known as black box testing, conformance-based, responsibility-based or functional testing) uses program specification criteria and seeks to establish conformance of the program under test to its requirements. This is achieved by constructing test cases that are sufficiently representative of the input domain and exercise the essential features of the program under test. Howden (1980) defines specification based testing as follows (Howden, 1980):

"In the “black box” testing approach to program testing, the internal structure of a program is ignored during test data selection. Tests are constructed from the functional properties of the program that are specified in the program’s requirements. The disadvantage of the black box testing approach is that it ignores important functional properties of the program which are part of its design or implementation and which are not described in the requirements." (p. 162).

According to Pressman (1997), Beizer (1995) and Patton (2001) specification-based testing targets the following types of errors:

i) Incorrect or missing program functions errors;
ii) Errors in the interfaces between various program functions;
iii) Errors in data structures or external database access;
iv) Input/output errors;
v) Performance errors; and
vi) Variable/program initialisation and termination errors.

The second category, program-based testing (also known as white box testing, implementation-based, fault-based- or structural testing) uses program implementation criteria and seeks to reveal implementation errors by constructing test cases that rely on
the program's source code. Program-based testing is defined as follows (Howden, 1980):

"Structural [program-based] testing is an approach in which the internal control structure of a program is used to guide the selection of test data. It is an attempt to take the internal functional properties of a program into account during test data generation and to avoid the limitations of black box functional testing." (p. 162).

Beizer (1990), Pressman (1997), and Patton (2001) suggest that program based testing attempts to ensure that:

i) All independent execution paths in the program have been exercised at least once;

ii) All logical decisions on their true and false sides have been exercised;

iii) All loops at their boundaries and within their operational bounds have been exercised; and

iv) All internal data structures have been exercised.

Program based testing helps uncover errors like data reference, declaration, and use errors, control flow errors etc. (Beizer, 1990; Patton, 2001).

While most of the surveyed sources draw a distinct line between specification- and program-based testing (Pressman, 1997; Schach, 1999; Sommerville, 1996), Binder (1994b) remarks that in object-oriented programming "The gap (and therefore usefulness) of the white/black-box distinction is decreasing." (p. 24). He attributes this to the fact that object-oriented programs are structurally different from procedural programs. The structural difference is based on the fact that, in general object-oriented methods contain less code than procedures in procedural programs. This reduces the scope for program-based testing (Jorgensen & Erickson, 1994).

In general, however, it is widely accepted that specification- and program-based testing techniques and methodologies complement each other (Binder, 2000; Pressman, 1997;

---

*Pressman (1997) defines a path as "a path through the program that introduces at least one new set of processing statements or a new condition." (p. 458).
Roper, Wood, & Miller, 1997). The following sections briefly discuss the most popular specification- and program-based testing techniques.

**Specification-Based Testing: Equivalence Partitioning**

Equivalence partitioning divides the input domain of the program under test into equivalent classes of data, from which the typical test cases to detect errors can be derived (Pressman, 1997). This suggests that, if a test case that belongs to a particular class is able to locate an error (or not), all test cases belonging to that class will also locate the same error (or not). The objective of equivalence partitioning is to reduce the input domain for test cases to a manageable size, which is an advantage of the technique (Roper, 1994). Roper (1994) also suggests that the effectiveness of the technique depends on the application domain. The main deficiency commonly attributed to the technique is that it does not take into consideration all types of test cases, such as high-yield test cases, for instance (Myers, 1979). Another deficiency is that it does not explore all combinations of input circumstances (Myers, 1979; Roper, 1994).

**Specification-Based Testing: Boundary Value Analysis**

Boundary value analysis is often considered to be an extension of equivalence partitioning and selects test cases at the boundaries of each equivalence class (see previous section) (Roper, 1994). The rationale behind boundary value analysis is that program errors normally occur while atypical test cases (i.e. the class boundary or edge partition values) are processed. The principal advantage of boundary value analysis is that it provides specific guidance for the selection of the test data. The major limitation of boundary value analysis is that it only works well if the program to be tested is a function of several independent variables that represent physical quantities (Jorgensen, 1995, 2002). Secondly, boundary values may have to be determined arbitrarily, if an equivalent class does not have explicit bounds. Finally, boundary value analysis cannot be applied to boolean type variables, because such variables only have two extreme values, namely true and false (Jorgensen, 1995, 2002).

**Specification-Based Testing: Cause-Effect Graphing**

Cause-effect graphing entails the identification of causes and effects from the software requirements artifact. Normally, software inputs correspond to causes, whereas, software outputs and transformations correspond to effects. The technique uses a
boolean logic network to link the causes to the effects (Pfleeger, 1991). The boolean logic network is then converted into a limited-entry decision table showing the effects occurring for each possible combination of causes and the test cases to be used to detect errors (Pfleeger, 1991). This technique has been found effective to identify omission and ambiguity errors in requirements artifacts (Roper, 1994). The first limitation is that cause-effect graphing does not explore boundary conditions, therefore, it cannot be used as a replacement for boundary value analysis; it should, however, be used to complement it (Pfleeger, 1991). Another limitation of cause-effect graphing is that its effectiveness is application domain dependent. For instance, Pfleeger (1991) argues that boolean logic networks are not practical for software that includes time details, iterations etc. Finally, given a large number of causes, the technique can become very complex. Consequently, users of the technique need to be highly experienced. However, despite the fact that the technique can be automated, it has not gained popularity (Pfleeger, 1991; Roper, 1994).

**Specification-Based Testing: State-Based Testing**

State-based testing is based on the use of finite state machines (FSM) which allow developers to model software behaviour (Beizer, 1990, 1995; Chow, 1978). An FSM models the behaviour of software in terms of a collection of states and state transitions caused by inputs (or stimuli) from the external environment of the software. During state transitions, software may produce observable outcomes (e.g. operations) that are commonly referred to as outputs. Beizer (1990) shows that state-based testing can uncover errors associated with unspecified and contradictory state transitions, unreachable and dead states etc. One problem that is frequently associated with state-based testing is that it is not easy to build FSMs for complex software, unless automatic tools are used (Jorgensen, 2002).

**Specification-Based Testing: Random Testing**

Random testing is used to test specific functions of the program (Howden, 1987). The technique is based on the use of a random number generator program that randomly selects test cases from the input domain. The effectiveness of random testing is controversial. Myers (1979) describes it as the poorest specification-based testing technique on the grounds that it uses no information from the requirements artifact. Others praise random testing as a valuable, simple, practical and often cost-effective test...
case generation technique (Duran & Ntafos, 1984; Leveson, 1991; Pizzarello, 1984). Some studies point out that random testing can be effective under conditions where the program to be tested is known to be error-prone or is at the early stages of its development (Loo & Tsai, 1988). Finally, in another study random testing was found to be effective in inferring about the operational reliability (i.e. the continuous delivery of correct functionality by software) of the program under test (Chen & Yu, 1996; Lyu, 1996).

**Program-Based Testing: Logic Coverage**

Logic coverage presupposes the use of program structure to generate test cases such that every statement, decision branch or condition, etc. is executed at least once. Chienski (1994) defines logic coverage criteria as a continuum of five levels ranging from the least to the most effective. The weakest logic coverage criterion is called statement coverage and requires that all statements be executed at least once. The next coverage criterion in the continuum is called decision coverage (also known as branch coverage) and it requires that every point of entry and exit in the program be invoked at least once and every decision in the program take all possible outcomes at least once. The literature suggests that statement coverage and decision coverage are commonly accepted as the minimum mandatory program-based testing requirement (Beizer, 1990). The strongest coverage criterion is called multiple condition coverage and it requires that every point of entry and exit in the program be invoked at least once, and that all possible combinations of the outcomes of the conditions within each decision be taken at least once. The complete list of logic coverage criteria can be found in Chienski & Miller, (1994); Roper, (1994). In general, the surveyed literature suggests that coverage criteria, depending on the choice of the criterion and the complexity of the program under test, can be very complex and time consuming. On the positive side, the literature survey has shown that logic coverage can be automated (Bertolino, Mirandola, & Peciola, 1997; Patton, 2001).

**Program-Based Testing: Data Flow Testing**

Data flow testing entails tracing of individual variables and their data through the software program (Patton, 2001). This differs from specification-based testing. While with specification-based testing a developer would know what the value of a variable is at the beginning and at the end of a procedure, with data flow testing, the developer
focuses on the intermediary values of the variable as well (Patton, 2001). Specifically, tracing a variable and its data includes knowing what data values a variable assumes (i.e. variable definitions), the circumstances where the variable and its datum (or data) is used (i.e. a predicate or computation use), and when the variable is killed (Jorgensen, 1995). Rapps (1985) and Beizer (1990) present a set of data flow testing techniques to uncover errors and organise them according to their relative strength in a subsumption hierarchy. The testing strategies differ in the extent to which predicate uses and/or computation uses of every definition of every variable in a program are exercised under some test (Beizer, 1990; Frankl & Weyuker, 1988; Rapps & Weyuker, 1985). Data flow testing is considered effective to uncover errors. However, data flow testing also requires lots of effort and is a tedious activity due to the enormous amount of bookkeeping involved. This disadvantage has, however, been offset by introduction of automated data flow testing tools (Beizer, 1990).

Program-Based Testing: Path-Oriented Testing
Path-oriented testing makes use of the control flow information of the program under test. Fundamentally, path-oriented testing consists of two major activities (White, 1987):

i) Select a path or a set of paths along which testing is to be conducted; and
ii) Select the test cases that will cause the selected paths to be executed.

An important advantage of path-oriented testing is the explicit association of the path and the input data that will trigger its execution. On the downside, complex programs may have a very large number of paths, and the test data generation and execution of each possible path may prove an impractical and expensive exercise (Beizer, 1990). Path oriented testing has been found effective to identify missing path errors and errors in conditional statements. Coincidental correctness is identified as the principal limitation of path-oriented testing (White, 1987; White & Cohen, 1980).

Table 2.5 summarises the above testing techniques according to the type and the main test case generation criterion.
As Table 2.5 suggests, testing techniques can be categorised into two groups, specification-based and program-based testing techniques. Specification-based testing techniques have in common the fact that they all view the program under test as a mathematical function that maps its inputs into outputs (Jorgensen, 2002). Individual specification-based testing techniques differ from each other in the degree of sophistication required to use a technique. Sophistication can be seen as the union of two effort components, namely, the judgemental and mechanical effort, both of which are required to be invested into by a developer in order to apply a testing technique.

Between the two effort components, it is the judgemental effort that is the most important and is proportional to the degree of sophistication of a testing technique. The mechanical effort component, however, while important for the practical use of a testing technique, does not significantly affect sophistication, because the more significant the mechanical component of a testing technique, the higher the potential for automation of the technique (Jorgensen, 2002).

Various testing techniques can be compared with each other by using sophistication as a comparison criterion. For example, both equivalence partitioning and boundary value analysis require input domain partitioning in order to generate test cases which complement each other. The only judgement that is required for both equivalence partitioning and boundary value analysis is the determination of the equivalence classes. After that, test case generation effort is mostly mechanical. Similarly, relatively more judgement effort is required for cause-effect graphing and state-based testing. Cause-effect graphing and state-based testing are conceptually similar. Causes and effects (in cause-effect graphing) are similar to inputs (or stimuli) and outputs (in state-based testing), but they differ in the way they are implemented. The former uses boolean logic networks and decision tables, whereas the latter uses finite state machines. Both require

<table>
<thead>
<tr>
<th>Technique</th>
<th>Type</th>
<th>Test Case Generation Criterion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Equivalence Partitioning</td>
<td>Specification-based</td>
<td>Input domain partitioning</td>
</tr>
<tr>
<td>Boundary Value Analysis</td>
<td>Specification-based</td>
<td>Input domain partitioning</td>
</tr>
<tr>
<td>Cause-Effect Graphing</td>
<td>Specification-based</td>
<td>Cause-effect relationships between input/outputs</td>
</tr>
<tr>
<td>Random Testing</td>
<td>Specification-based</td>
<td>Usability (Random)</td>
</tr>
<tr>
<td>Logic Coverage</td>
<td>Program-based</td>
<td>Logic Path Execution</td>
</tr>
<tr>
<td>Data Flow</td>
<td>Program-based</td>
<td>Data Definition-Use Path Execution</td>
</tr>
<tr>
<td>Path Oriented</td>
<td>Program-based</td>
<td>Control Path Execution</td>
</tr>
</tbody>
</table>
considerable judgement effort in order to consider data (i.e. causes/effects or inputs/outputs) and logical dependencies (causes versus effects or inputs versus outputs).

The sophistication of testing techniques is important because, according to Jorgensen (2002), it determines the number of test cases that are successful for detecting errors. In this context, Jorgensen (2002) suggests that the higher the sophistication of a testing technique, the more complete and minimal the number error-revealing test cases. This conclusion, however, appears to be based on limited empirical evidence, which suggests that more work is needed to evaluate the degree of sophistication of available testing techniques and its relationship to the number of error-revealing test cases.

Jorgensen (2002) shows that, in general, specification-based techniques suffer from two problems, namely gaps and redundancies. The gap problem arises when a testing technique generates test cases which test only some parts of the program and ignore others. For example, if a program has 10 possible paths of execution, a testing technique is said to suffer from a gap problem if it generates test cases that test only 5 of the 10 possible paths. A redundancy problem occurs when a testing technique generates test cases which test the same part of the program for the same errors more than once. For example, if a testing technique generates test cases A, B, and C which test only a single path in a given program and reveal the same errors, then the testing technique suffers from a redundancy problem. Jorgensen (2002) argues that in order to avoid gap and redundancy problems, the specification-based testing techniques should be used in combination with program-based testing techniques. Program-based testing techniques are similar to each other in that they all attempt to identify multilevel subsumption hierarchies of program coverage criteria. It is claimed that different levels of coverage do identify progressively more errors.

The combination of specification- and program-based testing techniques is referred to as a hybrid testing method and is highly recommended by Jorgensen (2002). Jorgensen's (2002) discussion, however, falls short of generalising rules or criteria about how to develop hybrid testing methods. Given the plethora of specification- and program-based testing techniques, a developer may find it rather difficult to determine which specification-based technique(s) should be combined with which program-based
technique(s). In addition, it is also important that developers know which types of errors would be detected by the various combinations of specification- and program-based techniques (i.e. hybrid testing methods). The fact that the answers to these questions were not found in literature suggests that further research is required in this direction.

2.2.4.2 Testing Different Levels of Abstraction

Software development normally progresses through different levels of abstraction. In the highest level of abstraction the problem is stated in abstract terms which are compliant with the application domain language. In the lowest level of abstraction the problem is stated in a language that can be directly implemented as an executable computer program (Pressman, 1997). Pressman (1997) cites Wasserman (1983) who argues the need for abstraction levels:

"The psychological notion of "abstraction" permits one to concentrate on a problem at some level of generalisation without regard to irrelevant low level details;" (p. 347).

In software development, levels of abstraction have a direct implication for testing, because they determine what will be tested for errors and how exactly testing must take place in order to expose the presence of errors. Consequently, the designation of levels of abstractions and their definition affects how errors can be identified, i.e. how testing techniques can be defined. While different development paradigms (e.g. procedural and object-oriented paradigms) are consistent with the way they designate levels of abstraction (e.g. both procedural and object-oriented paradigms recognise unit, integration, and system levels), they differ with respect to the definition of some of these levels (e.g. unit level) (Chen, Chen, & Chung, 1999; McGregor & Sykes, 1992; Myers, 1979; Pfeeger, 1998; Pressman, 1997; Smith & Robson, 1992; Sommerville, 1996). The following sections compare the different abstraction levels designated in the procedural and object oriented paradigms and the way these levels affect testing for errors.

Unit Testing

The key advantage of introducing the concept of units for testing purposes (in both procedural and object-oriented programs) is that small portions of a large program can be tested for errors in isolation, before the rest of the program is written and integrated.
Consequently, error hunting can be confined to smaller, more manageable chunks (Hunt, 1996).

There is a universal agreement in the literature that the class is the natural unit for the design of test cases for an object-oriented program (Berard, 1993; Binder, 2000; Jacobson, Christerson, Jonsson, & Overgaard, 1992; Yourdon, 1994; Chen et al., 1999; McGregor & Sykes, 1992; Smith & Robson, 1992). By definition, class testing takes care of the correctness of method interaction within a class and is the smallest executable unit (Binder, 2000; Chen et al., 1999). This differs from the procedural paradigm, where the smallest executable unit is the individual procedure (Jacobson et al., 1992; Myers, 1979; Pfleeger, 1998; Pressman, 1997; Sommerville, 1996).

Pressman (1997) argues that in procedural programs, unit testing (i.e. testing of procedures) is necessary because it helps uncover certain kinds of errors which may otherwise go undetected. The common errors detected via unit testing in procedural programs include 1) misunderstood or incorrect arithmetic precedence, 2) mixed mode operations, 3) incorrect initialisation, 4) precision inaccuracy, and 5) incorrect symbolic representation of an expression. In order to uncover these types of errors, developers need to use one or more of the testing techniques that are summarised in section 2.2.4.1.

In object-oriented programs, a method in a class is considered as a program written in a procedural language, and is, therefore, tested for the same errors as indicated by Pressman (1997) (i.e. like a procedural program unit). D'Souza & LeBlanc (1994) motivate the need to test methods in object-oriented programs as follows:

"But is it really necessary to test the routines [methods] individually? In other words, is it sufficient to test classes only at the cluster level and class level? Although the answer may seem affirmative, in actuality such testing may be quite inadequate. This is because such an approach to testing cannot guarantee statement adequacy for the class code. Consequently there will be some executable parts of the class code that may remain untested, leading to an overall lack of quality of class implementation. However, if routines [methods] are tested individually, there will be an assurance that all executable code has been tested to some degree, and there will be more confidence in its correctness." (p. 33).
In object-oriented programs, methods in a class interact with each other by modifying the set of shared variables (i.e. the instance variables of the class) of the object that encapsulates them (Alexander, 1999; Barbey, 1997; Barbey & Strohmeier, 1994b). The interactions between methods in a class must be tested for errors. Testing such interactions constitutes class testing in object-oriented programs, and is similar to the integration testing of procedural programs which is discussed later in this chapter (Harrold & Rothermel, 1994; Kim & Wu, 1996b; Kirani & Tsai, 1994).

Binder (2000) and D'Souza & LeBlanc (1994) argue that class testing is necessary and should be performed after method testing for the following reasons:

i) Many errors go undetected, if class testing is minimal or skipped altogether. This is because, ensuring the methods of a class function correctly by themselves does not guarantee that they will function correctly when used in conjunction with other methods in the same class; and

ii) If class level errors escape class testing, they will be much harder and expensive to correct when identified later in the development (i.e. during integration or system testing).

The survey of literature indicates that some class testing techniques have been developed by tailoring existing procedural testing techniques to suit an object-oriented class structure, while other class testing techniques have been developed anew. In general, four categories of class testing techniques have been identified. These are summarised in figure 2.4.

![Figure 2.4 - Class Testing Techniques](image-url)
The four categories of the testing techniques, as represented in the diagram in figure 2.4, are examined individually in the following subsections.

Class Testing Using Data Flow Testing Techniques
Most of the surveyed class data flow testing techniques are based on different software development artifacts (where the class is specified) in order to generate test cases. For example, Harrold & Rothermel's (1994) class data flow testing technique is based on the code of the class. Hong, Kwon, & Cha's (1995) technique is based on the diagrammatic representation of the class as a finite state machine. Kim et al. (1999) use a class UML state diagram to perform data flow testing. In these studies, specialised algorithms are used to transform the initial artifact into a flow graph automatically. A flow graph depicts the logical control of a program using specialised graphic notations. A flow graph allows variable definitions and uses (also known as def-use pairs) to be easily identified (Harrold & Rothermel, 1994; Hong, Kwon, & Cha, 1995; Kim, Hong, Bae, & Cha, 1999a).

At this point, data flow testing techniques (see section 2.2.4.1) can be directly applied to test the flow graph representation of the class. While the techniques proposed in Hong et al., (1995); Kim et al., (1999a) are specification-based and can be applied in the absence of class source code, the technique proposed in Harrold & Rothermel, (1994) is program-based. All techniques were found to be relatively effective in uncovering errors in a class and all techniques can be automated.

Boujarwah, Saleh, & Al-Dallal (2000) adopt a different approach to class testing using data flow techniques. They instrument the class code by manually inserting software probes whose objective is to collect information about the definitions and uses of the variables while the class under test is executing. After class execution, the collected information is tabulated and analysed against the class specifications, which is portrayed using an extended version of finite state machines. Boujarwah, Saleh, & Al-Dallal (2000) claim that this analysis helps detect possible class errors. While this approach was shown to be effective, its practicality may be questionable given that class instrumentation and result tabulation may prove to be quite tedious for large and complex applications. Besides, no indication was found that these steps can be automated (Boujarwah, Saleh, & Al-Dallal, 2000).
Many studies agree that an object-oriented class can be modelled as a finite state machine. In this context, Binder (2000) argues that:

"The packaging of instance variables and methods into a class is fundamental to object-oriented programming. The resulting interactions, dependencies, and constraints on message sequences are collectively called behaviour. Although, the number of message sequences and instance variable value combinations is infinite for practical purposes, a state machine can nevertheless provide a compact and predictable model of [class] behaviour." (p. 176).

Many studies have used class finite state machine models to develop test cases to identify errors. Examples include the studies presented in Gao, Kung, Hsia, Toyoshima, & Chen, (1995; Hoffman & Strooper, (1995, 1997); Kung et al., 1994; Turner & Robson, (1993, 1995). The models described in these studies have in common the fact that they analyse instances of a class (i.e. objects) as a collection of states which are determined by the values of class instance variables. These studies have developed variations of the test case generation algorithm developed by Chow (1978). The objective of such algorithms is to identify sequences of methods that cause erroneous class state transitions to occur.

Other studies use state-based testing concepts in a different manner (Battig, 1998; Binder, 2000; Firesmith, 1993; Jezequel & Meyer, 1997; Marick, 1995b; Rangarajan, 1999; Wang, King, & Wickburg, 1999). In these studies, a class is instrumented with statements\(^5\) which are commonly referred to as built-in test code or simply BIT. An exception is Rangarajan’s (1999) study where BIT is not embedded in the class itself, but instead in a special program external to the class, yet capable of accessing the encapsulated class state information. In any case, the objective of BIT is to ensure that instances of a class (i.e. objects) do not assume invalid states. Since class instrumentation with BIT entails the insertion of additional code in the class under test (which means that the class will be different from what was originally specified in the

---

\(^5\) Generally, such statements are categorised as of pre- and post-conditions and invariants (Meyer, 1997). Pre-conditions are assertions of what should be true before an algorithm starts. Post-conditions are assertions of what should be true after the algorithm ends.
class requirements), it is possible that new errors may be introduced. In addition, it is also possible that the instrumented code may interact with the code that provides the required functionality of the class and generate errors. Consequently, the use of BIT is controversial in some circles. Some authors regard built-in tests as intrusive to the ultimate purpose of software (Andersen, 1996; Rangarajan, 1999). Others have found built-in tests to decrease software performance and increase development costs (Battig, 1998; RST, 1994). Nevertheless, there are other researchers who have found BIT effective in increasing the error-revealing ability of a class, and therefore beneficial to the testing classes (Battig, 1998; Binder, 1994a).

In general, state-based testing techniques were found to be able to detect state-dependent errors (incorrect state, incorrect transitions, extra and missing state errors, etc.) that were impossible to detect with other testing techniques. This supports the recommendation that state-based testing is complementary to specification- and program-based techniques. However, the existing controversies also suggest that more empirical research is required in class state-based testing.

**Class Testing by Examining Pointers**

D’Souza & LeBlanc (1994) claim that using pointer aliasing information can be a powerful way of detecting errors in object-oriented classes. Their technique works as follows. The run-time structure of a class is examined and stored in a table of path names. Each path name is comprised of an object reference or pointer and the dynamic type of the object associated with that path name. Sorting the table reveals situations where two or more object pointers (i.e. corresponding to path names) point to the same object. These situations constitute anomalies and, therefore, suggest the presence of errors (D’Souza & J., 1994). The authors used contradiction to prove that this technique is as powerful as other techniques developed at that time. An advantage of the technique is that it discovers the causes rather than the symptoms of class errors. However, the disadvantage is that the information obtained by examining pointers can be too voluminous. Consequently, the practicality of the technique may be adversely affected.
Class Testing Based on Data Bindings

The objective of the technique presented in Kim & Wu, (1996b) is to test a class by considering varying sequences of data interactions between class methods. Such interactions are otherwise referred to as data bindings. The technique is based on the premise that class instance variables are shared among the methods defined in the class. Flow graphs are constructed for each individual instance variable and the methods that manipulate its data. Such flow graphs are called slice-flow graphs. It is claimed that testing all possible method call chains on a slice-flow graph will help recognize incorrect method sequences and generate error-revealing test cases (Jorgensen & Erickson, 1994). A method call chain constitutes a path that starts with a method that calls another method and so on. A path ends when a method is reached which does not make any calls of its own. Kim & Wu (1996b) claim that this methodology generates an increased number of mutually independent test cases faster and easier in comparison to other class testing techniques, but they do not specify what other class testing techniques are being referred to.

Integration Testing

Integration testing is indispensable in testing programs for errors because unit testing can never reveal integration-level errors (Jorgensen & Erickson, 1994). In this context, Pressman (1997) argues about the necessity and value of integration testing for procedural programs in that:

"The problem, of course, is "putting them [the procedures] together"-interfacing. Data can be lost across an interface; one module [procedure] can have an inadvertent, adverse effect on one another; subfunctions [procedures], when combined, may not produce the desired major function; individually acceptable imprecision may be magnified to unacceptable levels; global data structures can present problems -- sadly the list goes on and on." (pp. 498-499).

An identical argument is put forward by Binder (2000) for the integration of object-oriented classes where he suggests that integration testing reflects the relationship and interaction among a set of objects that interface with each other, assuming a correct behaviour for every single object involved (Chen et al., 1999; Overbeck, 1994; Tai &

---

*Such paths are also known as MM-paths (Jorgensen, 2002; Jorgensen & Erickson, 1994).*

It can, therefore, be concluded that integration errors would go undetected by just testing the program units (i.e. procedures or classes) in isolation. In the discussion that follows, the term unit will be used generically to refer to both procedures (in procedural programs) and classes (in object-oriented programs), when the discussion is applicable to both object-oriented and procedural programs. Otherwise, the designated terms (i.e. procedures and classes) will be used.

Binder's (2000) comprehensive work on object-oriented testing and works of a similar nature on procedural program testing (e.g. Beizer (1990); Sommerville, (1995); Pressman (1997)) show that systematic integration testing patterns have been developed and used since the early 1970s. Some of these integration patterns rely on the type of dependencies between units. For instance, one type of dependency is the client-server dependency which describes which unit uses other units (i.e. the client) and which unit is used by other units (i.e. the server).

Integration patterns that rely on dependencies that exist in both object-oriented and procedural programs are equally applicable to both types of programs. Consequently, such patterns may be equally effective to uncover integration errors in both object-oriented and procedural programs. For instance, client-server relationships exist in both object-oriented and procedural programs, therefore, integration patterns that rely on client-server relationships can be used to integrate test programs in both paradigms. Other integration patterns rely solely on object-oriented specific dependencies (e.g. aggregation, association, inheritance, etc.). In such cases, their applicability is confined to object-oriented programs. In this context, Jorgensen & Erickson (1994) distinguish between procedural and object-oriented integration testing by stating that:

"Because [in object-orientation] the concept of main program is minimised, there is no clearly defined integration structure. Thus [unlike in procedural]
systems] there is no decomposition tree to impose the question of integration testing order of objects." (p. 33).

The most popular and frequently cited integration patterns are summarised in the following sections.

**Big Bang Integration**

Big bang integration requires that all units be brought together without considering their dependencies or the risk that such integration may bring about (Binder, 2000). This type of integration may be the appropriate approach for monolithic procedural systems where it is impossible to separate tightly coupled units or for poorly designed object-oriented implementations (Foote & Yoder, 1997). Binder (2000) claims that big bang integration targets errors in an ambiguous and opportunistic way and is therefore, discouraged. In this context, Beizer (1984) says:

"In its purest (and vilest) form, big-bang testing is no method at all—'Let's fire it up and see if it works!' It doesn't of course." (p. 160).

Binder (2000) continues, by saying:

"A common result of [big bang integration] is that there are so many interface bugs [errors] that the system barely runs, defeating Big Bang Integration tests and making it hard to know where to look for and diagnose a failure." (p. 651).

Binder’s (2000) statement suggests that the principal drawback of big bang integration is error correction difficulty, due to the fact that developers are likely to have only few indications about the location of errors, and hence are unable to easily correct them.

**Bottom-Up Integration**

Bottom-up integration testing requires the development of a dependency hierarchy showing which units are clients (i.e. use other units) and which are servers (i.e. are used by other units). The servers which are located at the bottom of the dependency hierarchy are tested first. Generally, complete testing of a server requires that its interface with the client be tested for possible errors. In this case, a driver is developed. A driver is a simplified throwaway version of a client, whose objective is to temporarily imitate and exercise the actual interface between the actual client and the server. When errors are
identified and removed from the server, software developers work their way up the dependency hierarchy and iteratively replace drivers with actual units until all units are integrated into the working software. Binder (2000) and Jorgensen (1995) suggest that driver development is the most significant cost of bottom-up integration. Such cost, however, may be offset if drivers are designed to be reusable (Gamma & Beck, 1998). In general, the effectiveness of bottom-up integration to uncover integration errors has been played down in some studies (Solheim & Rowland, 1993), and confirmed in others (Jakobsen, 1998).

**Top-Down Integration**

Like bottom-up integration, top-down integration requires the development of a dependency hierarchy showing clients and servers. Unlike bottom-up integration, units at the top of the dependency hierarchy, i.e., the clients, are tested first. A client that is being tested for errors may need to use a server that has not been tested yet. In this case, a dummy server is written to temporarily replace the actual server. The dummy server is also known as a stub. A stub has the same interface as the unit it is supposed to substitute and has a very limited functionality (Pfleeger, 1991; Sommerville, 1996). The stub is used until all errors in the client have been identified and corrected. Client testing is repeated across the different levels of the dependency hierarchy by substituting stubs with full functionality units, and stubbing lower level servers (Binder, 2000). Reaching the servers at the bottom of the dependency hierarchy signals the completion of top-down integration. Binder (2000) and Sommerville (1996) suggest that top-down integration helps discover unnoticed design errors early in system development, while providing early demonstrations of end-to-end system functionality. The cost of driver development in top-down integration is minimal, but it is offset by the cost of stub development. Solheim & Rowland (1993) studied different integration testing patterns with respect to their error detection ability. They found that top-down integration had a higher error detection rate, which they attribute to the fact that top-down integration patterns exercise more components per test case than other patterns do. Finally, the literature suggests that there is more throwaway code in top-down integration than bottom-up integration, implying that top-down integration is more costly (Jorgensen, 1995).
Object-Oriented Integration

In recent years, the integration of object-oriented classes appears to have received special attention in literature. This is attributed to the fact that object-oriented classes are subject to unique kinds of relationships, namely, inheritance, aggregation, and association. These types of relationships may contain errors, which must be detected during the integration of the classes (Alexander, 1999a).

A seminal study on object-oriented integration testing was carried out by Harrold, McGregor & Fitzpatrick (1992). This study focused on error identification between classes related by inheritance. The proposed technique incrementally updates parent class testing information to reflect modified, inherited and newly defined elements (e.g. instance variables and/or methods) in the subclasses (Harrold, McGregor, & Fitzpatrick, 1992). This work has been extended by Alexander & Offutt (1999) where errors resulting from polymorphic relationships between classes in inheritance hierarchies are considered. These errors are identified using a hierarchy of coupling criteria that formalises the different levels of dependency that can exist between classes in object-oriented programs (Alexander, 1999, 1999a; Jin & Offutt, 1995, 1997, 1998).

Several other studies have aimed at determining the order in which classes should be tested and integrated in object-oriented programs (Andersen, 1996; Chen et al., 1999; Jeron, Jezequel, Traon, & Morel, 1999; Kung, Gao, Hsia, Toyoshima, & Chen, 1995d; Kung, Gao, Hsia, Lin, & Toyoshima, 1995a; Labiche, Thevenod-Fosse, Waeselynck, & Durand, 2000; Tai & Daniels, 1999). All these studies have in common the fact that they all use the inheritance, aggregation and association relationships as criteria to determine the order in which classes are tested for errors and integrated. In addition, the surveyed studies attempt to minimise the number of stubs and drivers that need to be developed to help testing and to minimise cost. These studies differ in the algorithms used to identify class integration order.

System Testing

According to Binder (2000) the scope of system testing comprises a complete integrated application. The test cases derived using system testing techniques focus on finding errors with functional (i.e. conformance to requirements) and non-functional (i.e. performance, stress, security, and load) capabilities (Humphrey, 1989a). System testing
of object-oriented programs has the same goal as system testing of procedural programs, i.e. to ensure that the collaboration of the individual units (i.e. procedures or classes) solves the target problem correctly and consistently without errors (Barbey, 1997; McGregor, 1999a; McGregor & Sykes, 1992). Consequently, there is consensus in the literature that system testing in both procedural and object-oriented programs, is essentially the same (Binder, 2000; Duncan, Robson, & Munro, 1999; Irvine & Offutt, 1995; McGregor, 1998b). Two system testing techniques appear to be most popular in literature. These techniques are 1) use case testing, and 2) operational profile testing.

Use Case Testing
Use case testing appears to be a popular system testing technique (Binder, 2000; Jacobson, Booch, & Rumbaugh, 1999; Jacobson et al., 1992). Jacobson et al. (1992) define a use case as follows:

"A use case is a specific way of using the system by performing some part of the functionality. Each use case constitutes a complete course of events initiated by an actor and it specifies the interaction that takes place between an actor and the system. A use case is thus a special sequence of related transactions performed by an actor and the system in a dialogue." (p. 159).

Binder (2000) describes the details of the use case testing technique. In essence, use case testing comprises three main steps, namely, use case development, test case development, and execution of use cases using the appropriate test cases. In general, Binder (2000) suggests that use case testing is good at uncovering the following types of errors:

i) domain, logic, and incorrect data handling errors,
ii) incorrect or missing dependency on system states established by prior use cases,
iii) undesirable system feature interactions, and
iv) omitted or not required system capabilities, etc.

The principal problem with use case development and testing is that there are no generally accepted guidelines that indicate the correct degree of specificity of a use case. This directly affects the quality of the test cases generated and therefore, whether such test cases will successfully uncover all errors.
Operational Profile Testing

Operational profile testing is driven by the motivation that the frequency of failures (caused by errors in a system) experienced by the user is proportional to the frequency with which different operations of the system are used. The latter is referred to as operational usage frequency (Binder, 2000; Musa, 1993a). In this context, Musa (1993) says that:

"Testing driven by an operational profile is very efficient because it identifies failures (and hence the faults [errors] causing them) on average, in order of how often they occur. This approach rapidly increases reliability-reduces failure intensity-per unit of execution time because the failures that occur most frequently are caused by the faulty operations used more frequently. Users will also detect failures in order of their frequency, if they have not already been found in test." (p. 28)

This type of testing has been found to be well-suited to test both procedural and object-oriented programs (Binder, 2000; Collofello, 1988). Operational profile testing is also called statistical testing (Walton & Poore, 2000; Walton, Poore, & Trammell, 1995).

Binder (2000) suggests a generic approach to conduct operational profile testing where the tester is supposed to prepare an operational profile of the system under test. Typically, an operational profile ranks system use cases and operations on the basis of their relative frequency of usage. The testing resources are allotted to use case testing according to their relative usage frequency. This is considered an advantage given testing budgetary constraints (Binder, 2000; Walton et al., 1995). The main drawback associated with operational profile testing is the relative difficulty in estimating accurately operational profiles for the system under test. Despite this, evidence of the effectiveness of operational profile testing has been reported in several experience reports, which can be found in Chang, Liao, Seidman, & Chapman, (1998); Musa, (1998); Runeson & Regnell, (1998); and Wohlin & Runeson, (1994).
2.2.4.3 Summary and Evaluation of Testing

Testing is commonly referred to as the execution of a program with the intent to find errors. The discussion about testing was made from two perspectives. First, testing techniques were categorised on the basis of the criteria used to generate test cases, which help expose errors. Second, testing techniques were categorised on the basis of the levels of abstraction that software development normally goes through. Where possible, comparisons between testing techniques applied to the two most popular development paradigms (procedural and object-oriented) were made.

Clearly, there exists a plethora of testing techniques that can be used to identify errors at different levels of abstraction of software using different criteria. Commonly, these techniques are detective in nature and despite the fact that they may be based on different development artifacts (e.g. design artifacts) to generate test cases, they are all meant to identify errors in executable artifacts (i.e. code artifacts). In addition, there are many testing techniques, all of which are different and are specialised to target different types of errors. This implies that there is no such thing as the best technique. Existing research suggests that some testing techniques complement each other, and consequently, they should be applied in combination with each other. For example, specification-based techniques should be combined with program-based techniques in order to test program units. Once tested individually, the interactions among units must be tested using integration testing patterns. Finally the system represented by the developed software must be tested as well using system testing techniques.

There is widespread agreement in literature that, when properly combined, testing techniques may be able to detect most errors in code artifacts (Jorgensen, 2002). Nevertheless, existing research does not seem to have systematically addressed issues concerning which combinations of testing techniques are best at uncovering different types of errors. In addition, many of the surveyed testing techniques are fairly sophisticated and complex. As a result, successful application may require experience, training, and other resources. Consequently, applying many complementary testing techniques may be costly. The cost factor may be a deterrent to their use in practice. Besides, the fact that testing techniques focus mainly on code artifacts means that although they may be successful in detecting errors, additional work may be still required in removing the detected errors. This may be particularly expensive for errors.
injected during the early phases of software development, i.e. pre-code artifacts (Hevner, 1997).

2.2.5 Summary and Evaluation of Error Detection Approaches
In section 2.2, a variety of error detection approaches were discussed. Error detection presupposes that the errors have already been injected into a development artifact and it is the developer's task to identify and remove them. As pointed out in the beginning of section 2.2, two broad categories of error detection approaches are recognised, namely execution- and non-execution-based. Initially, in section 2.2.1 and 2.2.2 reviews and inspection techniques were covered. In section 2.2.3, the use of metrics to detect error-prone sections of development artifact was discussed. These error detection approaches are non-execution-based. In section 2.2.4, the different approaches to testing software were explained. Testing approaches are execution-based.

The survey of literature has identified some studies, which have attempted to compare techniques from execution and non-execution based categories. For example, Roper (1994) cites the results of some studies carried out by Basili & Selby (1987) and Lauterbach & Randall (1989) where code reading, specification- and program-based testing techniques are compared. In general, the findings are consistent in the conclusion that code reading is more effective in detecting errors than both specification- and program-based testing.

The findings of the experiment conducted by Roper, Wood, & Miller (1997) comparing and contrasting between code reviews, specification- and program-based testing, are interesting, because they conclude that the relative strengths of a testing technique depends on the nature of the program and its errors. This suggests that an additional activity that developers need to perform before embarking on error detection, is to choose an error detection approach that best suits their application. No systematic work appears to have been carried out in this area, implying that making an informed decision about what error detection approach to use might turn out to be an expensive proposition. Roper, Wood, & Miller (1997) also found that the effectiveness of a technique depends on the human component, because different subjects using the same technique did not identify the same errors (Roper et al., 1997). This suggests that the
background of the programmer and the programmer's knowledge about errors have a
direct effect on the effectiveness of a testing technique.

There is agreement in literature that knowledge about errors is important for both
execution- and non-execution-based error detection approaches (Ciolkowski, 1999;
Laitenberger et al., 1999a; Lanubile et al., 1998; Overbeck, 1994). The work of
Lanubile et al. (1998), however, is the only one that suggests that knowledge about
errors can be accumulated using error abstraction. While the idea of error abstraction
was shown to be effective, it fails to specify what exactly should be abstracted about
errors and why. This needs further investigation.

The survey of literature shows that there exist many error detection approaches, most of
which require experience and automation (for testing techniques), and while these
approaches may be effective, they can also be expensive and time consuming to apply.
In addition, error detection approaches are limited to error detection only. This means
that error detection is a way to assess the quality of an artifact, not a remedy by itself
(Hendrickson, 2001). In order to obtain an error free artifact, additional work is required
to remove the detected errors from the artifact. This additional error removal work may
vary depending on which phase of the development errors were injected and how far
from that point they were detected. There exists significant evidence in the literature
that the farther an error is from its injection point, the more expensive that error is to
correct (Boehm, 1981, 1987a; Boehm & Basili, 2001). Although, error detection and
correction increase the amount of work by developers, it should be stipulated that error
detection and correction still do not guarantee error-free artifacts and software. This is
because the correction of a detected error may actually introduce new errors. This
requires regression testing to detect errors that may have been introduced during the
correction of previously detected errors (Leung, 1995; Rothermel & Harrold, 1996).
This suggests that producing error-free software artifacts using error detection may be a
lengthy and costly process.

The following section examines what error prevention approaches can offer. Error
prevention means that software artifacts are constructed in a way that error injection is
averted. The next section shows that less research has addressed error prevention as
opposed to error detection. However, the reviewed work suggests that error prevention
relies heavily on error detection in two ways. First, it relies on knowledge about the errors that error detection has generated. Second, it relies on the use of error detection approaches as soon as artifacts or their parts have been developed.

2.3 Error Prevention Approaches

As stated in the beginning of this chapter, the aim of error prevention is to prevent errors from being introduced into a software artifact. Jones (1985) suggests that error prevention is about “do[ing] it [development artifact] right the first time” (p. 150). Figure 2.3 (shown in the beginning of this chapter) shows that research on error prevention can be classified into four categories. Firstly, significant work has been done to prevent error injection into requirements artifacts. This work is discussed in section 2.3.1. Secondly, several studies have focused on the analysis of the causes leading to error introduction as the basis for error prevention. These studies are summarised in section 2.3.2. Thirdly, section 2.3.3 presents an error prevention methodology called Cleanroom software development. Finally, section 2.3.4 discusses some studies that use metrics to prevent errors.

2.3.1 Preventing Errors in Requirements Artifacts

Many studies consider requirements artifacts critical to the entire software development exercise (Lamsweerde, 2000). This is because other development artifacts (e.g. design, code etc.) rely on requirements artifacts and any errors injected and not removed from the requirements artifacts are likely to be carried over to subsequent development artifacts. The literature survey suggests that there are different approaches to prevent errors from being injected into requirements artifacts. A summary of the surveyed approaches is provided in figure 2.5 and each of these are discussed in turn in the following sections.
2.3.1.1 Goal Analysis

Lamsweerde (2000) argues that requirements errors can be avoided by using goal analysis. A goal is an objective that must be achieved by the software under consideration (Lamsweerde, 2000, 2001; Letier & Lamsweerde, 2002). The rationale behind goal analysis is that requirements are refinements of the system goals and "are sufficient to establish the goals they are refining." (Lamsweerde, 2000, p.7) (Anton & Potts, 1998; Lamsweerde & Letier, 1998). The principal advantage of incorporating goals into the requirements development is that goals provide a criterion for sufficient completeness of a requirements specification artifact (Lamsweerde, 2001). Consequently, software goal identification is the first step towards avoiding omission errors (Mylopoulos, Chung, & Yu, 1999). Software omission errors occur when necessary information is omitted from requirements artifacts. Second, goals help identify not only all functional requirements, but also the non-functional ones (Leveson, 1995; Mylopoulos et al., 1999; Nixon, 1993). Third, goal representation greatly facilitates the early phases of software development and provides a rationale for the stakeholders of the software supporting the proposed requirements (Lamsweerde, 2001; Lamsweerde & Letier, 1998; Mylopoulos et al., 1999). Fourth, incorporating goals into the requirements specification helps to avoid irrelevant requirements, which are frequently considered as deficiencies of requirements artifacts (e.g. extraneous information error) (Lamsweerde, 2001).

---

7 Functional requirements lead to particular functions or services that the system is expected to deliver (Lamsweerde, 2001). Functional requirements are stated formally and are enforced during the implementation (Mylopoulos et al., 1999).

8 Non-functional requirements are global qualities of the proposed software such as flexibility, maintainability, usability, extensibility, performance, security, safety, etc (Lamsweerde, 2001). Non-functional requirements are normally stated informally and are often contentious. For example, performance goals usually interfere with flexibility goals (Mylopoulos et al., 1999).
2.3.1.2 Viewpoint Analysis

Lamsweerde (2000) suggests that another way to prevent requirements errors from being introduced into requirements artifacts is by using viewpoints (Kaindl, 1995; Potts, 1995; Potts, Takahashi, & Anton, 1994). Viewpoints constitute all end-users or other systems that are interfaced to the software whose requirements are being specified (Kotonya & Sommerville, 1996). For example, in the ATM system described by Kotonya & Sommerville (1996) some of the identified viewpoints include the bank manager, the home customer, the foreign customer, security officer, etc. Lamsweerde (2000) suggests that viewpoint consideration helps point out possible inconsistency errors in requirements. Inconsistency errors exist if information in one part of a requirements artifact contradicts information in other parts. However, Kotonya (1996) criticises viewpoint analysis by suggesting that it lends itself to the possibility of individual viewpoints being considered in isolation from other viewpoints. Therefore, viewpoint analysis lacks the ability to depict possible interactions between viewpoints. For example, in a banking software application the interaction between a bank customer viewpoint and a bank manager viewpoint, in which the manager allows the customer to overdraw his account, may not be very clear.

Viewpoint analysis is similar to perspective-based reading (see section 2.2.2.3) because different perspectives are involved. The difference consists of the fact that in viewpoint analysis different perspectives are employed to construct an artifact, whereas in perspective-based reading different perspectives are employed to review an artifact that has already been constructed (not necessarily by using different viewpoints).

2.3.1.3 Use Case Analysis

Another type of analysis that can be used to avoid requirements errors is called use case analysis. Use case analysis is also known as scenario analysis. A use case constitutes a chronological sequence of interactions, events and exceptions, which result when the system collaborates with one or more of its users (see section 2.2.4.2). Potts (1995) suggests that one important advantage of use case analysis is that it can illuminate, before implementation, how a proposed system is likely to affect its actual users in different usage situations. Therefore, this may help avoid requirements errors. Use cases are also said to be helpful for validation purposes (Fickas, Karat, Johnson, & Potts, 1994; McGregor, 1998a). However, one problem inherent in use cases relates to the
likelihood of combinatorial explosions when all possible behaviours are enumerated (Lamsweerde, 2000). Consequently, use case analysis for complex software may be costly and time consuming, unless automated. Lamsweerde (2000) also suggests that use cases are procedural in nature and this may lead to overspecification which can be considered a disadvantage.

2.2.1.4 Prototyping

Prototyping can also be used to prevent requirements errors (Davis et al., 1993b). A prototype presents the users with an inexpensive and executable imitation of the required software where they can see directly whether the requirements provided by the prototype correctly fulfil their needs (Mason & Carey, 1983; Saiedian & Dale, 2000). This helps developers to get users to identify any possible requirements errors that might have been injected in the requirements artifacts early. Prototype development has been found particularly helpful in avoiding incorrect fact errors in interactive systems (Mason & Carey, 1983). Incorrect fact errors exist when information in the artifact conflicts with general domain knowledge or with what the user expects to be delivered by the software. While there is growing consensus that prototype development is a critical prerequisite to specify correct requirements, it must be added that prototype development is also expensive, because prototypes are meant to be disposable samples of the required software (Andriole, 1994; Mason & Carey, 1983; Saiedian & Dale, 2000; Sommerville, 1996). Another danger is that sometimes developers choose to evolve a 'good' prototype into the required software, allowing less-than-ideal solutions to be delivered (Pressman, 1997). An extensive discussion of different prototype development techniques and their advantages and disadvantages can be found in Carey & Mason, (1983) and Urban, (1992).

2.3.1.5 Formal Methods

Formal methods constitute an alternative way to prevent requirements errors. Formal methods use applied mathematics, typically formal logic, to develop requirements artifacts. Generally, formal methods describe a system as a mathematical model and have been found very effective in preventing all types of requirements errors (Ciolkowski, 1999). The literature suggests that there exist several formal methods to prevent requirements errors (Easterbrook et al., 1998; Greenspan, Mylopoulos, & Borgida, 1994; Heitmeyer, Kirby, & Labaw, 1997; Heitmeyer, Jeffords, & Labaw,
Despite the effectiveness of these formal methods to prevent requirements errors, they suffer from several disadvantages which have limited their application in the industry. First, formal methods are costly because: a) they are difficult to learn and understand, b) they are tedious and time-consuming, and, c) they require training of developers. Second, requirements specification artifacts developed using formal methods are highly technical and therefore difficult for users to read especially if they typically work in non-technical domains. Third, the application of formal methods is application domain-dependent. For instance, formal methods have been found easily applicable and very effective to prevent requirements errors in embedded software. However, formal methods appear to be very complex to apply in information systems software performing data transformations (Ciolkowski, 1999).

2.3.1.6 Summary
In section 2.3.1 approaches preventing errors from being injected into requirements artifacts have been discussed. Some of the approaches are specialised and prevent certain types of requirements errors (e.g. goals analysis is cited to be good at avoiding omission errors, etc.). During the application of such approaches, the prevention of other types of errors may occur, but this does not appear to be systematic. This implies that these approaches may be more effective if used in conjunction with each other. Formal methods appear to be an exception, and help to prevent most types of errors. Yet, they suffer from several disadvantages which make them impractical and rather difficult to use. This may explain the lack of popularity of formal methods in general in the industry.

2.3.2 Error Prevention Using Error Causal Analysis
Many studies have been carried out in prestigious companies, such as IBM, NEC, Computer Sciences Corporation (CSC) and Lucent Technologies with respect to error prevention (Card, 1998; Gale, Tirso, & Burchfield, 1990; Kajihara, Amamiya, & Saya, 1993; Leszak, Parry, & Stoll, 2000; Mays et al., 1990). These have been reported over a 10 year period (1990-2000) and propose team-based error cause analysis in order to prevent errors from being introduced into development artifacts. The surveyed studies are conceptually similar, yet there are minor differences in the way they are implemented.
First, the common cornerstone of the surveyed error prevention approaches is causal analysis. Causal analysis is commonly carried out in meetings, where the possible cause or causes leading to shortlisted errors⁶ are identified, discussed and agreed upon by a group of developers. Typically, error cause analysis meetings are carried out at the completion of a development phase. Causal analysis involves the abstraction of errors in generic terms and their classification into categories of error causes. The meeting also determines the prevention actions (also referred to as remedial actions, countermeasures etc.) and development phase changes to avoid similar errors from recurring in the future. In all studies, the aim of causal analysis and the generation of preventative actions is to achieve what Mays et al. (1990) refer to as "defect [error] extinction"—which means removing all existing errors from an artifact and preventing their future occurrences.

Second, another common element among the surveyed studies is the employment of designated action teams with the objective of ensuring that the preventive actions suggested in the cause analysis meetings are actually implemented. In all studies, the information about errors, error causes, preventive actions, and preventive action implementation results obtained by the action teams, is systematically recorded into indexed repositories which are publicly accessible within the organisation.

Third, the dissemination of error information in the repositories (i.e. information including error causes, preventive actions, and preventive action implementation results etc.) appears to be a common activity in all studies. However, in different organisations, this is implemented in different ways. Mays et al. (1990) use systematic kickoff meetings, which are conducted at the beginning of a development phase to prepare developers for the work of the phase. Kajihara et al. (1993) disseminate information through training sessions, seminars etc. In both cases, the information that is accumulated in the repositories is discussed among developers who are about to embark on a development phase or project, with particular focus on common error lists. Common error lists constitute compilations of errors that are expected to occur frequently during a development phase or entire software project. Both Card (1998) and Leszak et al. (2000) recognise the importance of the dissemination of the accumulated

⁶Shortlisted errors are identified using error detection techniques.
error information (i.e. the information in the repositories), but have not shown how exactly this dissemination must occur.

Error prevention using causal analysis has been found to be very effective in reducing the number of errors committed during the various phases of software development. Empirical evidence supporting the error prevention capability and productivity is provided in Card, (1998); Gale et al., (1990); Kajihara et al., (1993); Leszak et al., (2000) and Mays et al., (1990). Some drawbacks, however, have been identified. First, the fact that causal analysis to discuss errors injected in the artifact, takes place at the end of a development phase, presumes that error detection approaches (see section 2.2) must have been applied to the artifact prior to the error cause analysis meetings. In all studies, the use of such techniques is only alluded to. This suggests that error detection techniques play an important role in effectiveness of error prevention using causal analysis

The surveyed studies are inconsistent in the proposed error cause\textsuperscript{10} categorisation schemes. This may be attributed to the different nature of the software projects where the shortlisted errors are found, organisational standards, practices, etc. In addition, the cost of the resources required to administer error prevention, as suggested above, is quite significant, despite the fact that such cost is offset by the rewards resulting from the implementation of cause-based error prevention (Chillarege et al., 1992). Last but not least, these studies are commonly criticised in relation to their potential subjectivity, due to total reliance on qualitative opinions. Such opinions appear to be solely based on the human investigative capabilities of the developers who get involved in the causal analysis meetings to identify causes and attribute them to errors (Chillarege et al., 1992; Fredericks & Basili, 1998).

To address the issue of subjectivity related to the error prevention methodologies discussed above, Chillarege et al. (1992) have proposed an Orthogonal Defect Classification (ODC) methodology. The ODC methodology does not rely on developers to speculate possible causes of errors, rather it uses an error categorisation scheme

\textsuperscript{10} A detailed discussion of the different error cause classification schemes is provided in chapter 6.
which provides definitive easily measurable answers about error causes. In this context, Chillarege et al. (1992) state that:

"The goal of ODC is to provide an in-process measurement paradigm to extracting key information from defects [errors] and enable the metering of cause-effect relationships." (p. 944).

Chillarege et al. (1992) add that:

"ODC essentially means that we categorize a defect [an error] into classes that collectively point to the part of the process that needs attention, much like characterizing a point in a Cartesian system of orthogonal axes by its (x, y, z) coordinates." (p. 945)

ODC is based on two error attributes. First, the error type attribute, which characterises what needs to be corrected, and can be linked to different phases of the software development. Normally, errors injected in different phases of development, bear their own signature in their distribution for the phase where such errors are corrected, if such errors are charted using ODC's error type attribute. For simplicity, such distributions of signatures are called baseline distribution signatures. At the completion of each development phase, the number of different corrected errors categorised by the error type attribute is charted. Any departure from baseline distribution signature points to a deficient phase in the development that may be causing errors. Consequently, further attention may be paid to that phase. This is known software development process inferencing.

Second, the error trigger attribute constitutes a condition that can lead to the exposure of errors in an artifact. Chillarege et al. (1992) justify the need for and the usefulness of error triggers thus:

"The concept of the trigger provides insight not on the development process directly, but on the verification process. Ideally, the defect trigger distribution for field defects [errors] should be similar to the defect [error] trigger.

11 In the original ODC publications, the term defect has been used (Chillarege et al., 1992). In the beginning of this thesis, it was stated that the term error would be used throughout the thesis to represent errors, defects, and faults. In order to main consistency, in this thesis, the term error will be used to refer to Chillarege et. al.'s ODC defect term.
distribution found during system test. If there is a significant discrepancy
between the two distributions, it identifies potential holes in the system test
environment."

(p. 950).

Therefore, error triggers measure the completeness and the effectiveness of the artifact
error detection activities, such as testing, reviews, inspections etc., and reiterate the fact
that error detection and error prevention are closely related.

While many studies attest the effectiveness of ODC in identifying error-causing
development phases, there are some ODC implementation issues that are not very clear.
For instance, it is not clear in what way baseline distribution signatures are determined.
Also, corollary questions such as, the factors (e.g. project application domain, project
scope, experience of developers) determining the shape of baseline distribution
signatures are left unanswered.

Yu (1998) at Lucent Technologies takes a different approach to causal-analysis error
prevention. He reports that a team at Lucent Technologies have developed a catalogue
of C code errors commonly encountered during the coding of switching systems. The
development of the catalogue of C errors was based on interviews conducted with
various developers about common errors, their root causes and possible prevention
guidelines which would avoid their future introduction into code artifacts. The catalogue
of C errors was used by the developers as a training and reference guide in the
development of subsequent releases of switching software. Yu (1998) reports that the
information provided by the catalogue of C code errors helped reduce the risk of
introducing any of the catalogued errors into code artifacts, while reducing the cost of
testing and error correction. While Yu’s (1998) study indicates the effectiveness of a
catalogue of C code errors, it also raises many other questions for further research. For
instance, it raises the question of whether a similar catalogue would be equally effective
to help prevent errors in pre-coding phases of software development. The errors in Yu’s
(1998) catalogue have been described in terms of causes and prevention guidelines. This
raises the question of what other error related information might help describe errors
better. In addition, Yu’s (1998) catalogue is language-dependent and this fact raises the
question of whether similar catalogues would be equally effective in different languages
and development environments (e.g. Java, etc.). Finally, Yu’s (1998) study did not
indicate whether the catalogue of C errors has any effect on the productivity of developers. Clearly, such issues require investigation and they are addressed in this thesis.

2.3.3 Cleanroom Software Development

Cleanroom software development focuses on error prevention rather than error detection and correction, and has a primary objective of the development of software that exhibits no failures in use (Linger & Trammell, 1996). Linger & Trammell (1996) capture the essence of Cleanroom software development processes as follows:

"Key characteristics of the Cleanroom process are the incremental development lifecycle and independent quality assessment through statistical testing. The development life cycle starts with a specification that not only defines function and performance requirements, but also identifies operational usage of the software and a nested sequence of user-function subsets that can be developed and tested as increments which accumulate into the final system. Disciplined software engineering methods provide design and verification techniques required to correct software. Correctness verification by development teams is used to identify and eliminate defects prior to any execution of the software."

(Linger & Trammell, 1996).

Linger & Trammell's (1996) statement reiterates the three principal elements of Cleanroom software development which are carried out in a team effort using:

i) an incremental development model;

ii) a box structure specification and design, correctness verification and statistical testing; and

iii) ongoing reviews.

Incremental development models allow development stability, while providing users with the opportunity for change. Incremental development allows the functionality of software to be partitioned into a series of increments, which are developed and delivered one by one. In Cleanroom software development, each increment is formally specified and while it is being developed, its specification is frozen. This, however, does not apply to the specification of remaining increments allowing users to change them (Sommerville, 1996, 1996a).
Cleanroom software development uses a box structure for requirements specification and design representation. The box structure comprises three distinct forms of system representation, known as the black box, the state box, and the clear box. The black box structure maps only external, user-observable box structure stimuli (inputs) into software responses (outputs). The state box structure is derived and verified against a corresponding black box and it encapsulates state-dependent data, in terms of data structures. The state box maps each stimulus and an existing state into a system response and a new resulting state (also known as state transition). The clear box structure contains the design of the procedures required to implement the state box transitions.

When the system is specified and designed using these three box structures, its correctness is verified. Correctness verification is performed using formal mathematical proof techniques. While correctness verification has been found remarkably effective in eliminating errors early (Linger & Trammell, 1996), it is also considered quite expensive, difficult, and requires significant developer training. In this context, Trammell, Binder, & Snyder (1992) argue that:

"Although practitioners learn formal mathematical proof techniques in Cleanroom training, a balance of formality and economy of effort is emphasized in practice" (p. 87).

This implies that mathematical proof techniques may not always be economically justified.

The goal of statistical testing, also known as operational profile testing (see above section 2.2.4.2, Operational Profile Testing), is the measurement and certification of software reliability rather than the identification of errors per se. Software reliability is measured by observing the number of failures caused by errors that occur during the execution of probable operational profiles (Sommerville, 1996). For example, if the maximum number of allowable failures is exceeded, redesign is warranted.

Reviews are checklist-driven (see above section 2.2.1 and 2.2.2) structured activities involving the author of the artifact under review and at least another developer who has
not seen the artifact before. Hence, they are also referred to as peer reviews. Peer reviews are organised in order to detect any errors that might have been injected into design and code artifacts and to check whether standards have been adhered to.

Many success stories highlighting the effectiveness of Cleanroom software development have been reported in literature (Basili & Green, 1994; Hausler, Linger, & Trammell, 1994; Krasner, Terrel, Linchak, Arnold, & Ett, 1992; Linger, 1993, 1994; Linger & Trammell, 1996; Oshana & Linger, 1999; Sommerville, 1996, 1996a; Trammell, Binder, & Snyder, 1992). These studies have commonly and consistently concluded that Cleanroom software development helps improve the quality of the delivered software in terms of significant reductions in error rates, significant reductions in system failure rate and the minimisation or total elimination of design and code compilation errors. In addition, Linger & Trammell (1996) suggest that other improvements have been noted in productivity, and return on investment. Also, Linger & Trammell (1996) cite Ett & Trammell (1996) noting that the Cleanroom software development does not recognise paradigm boundaries, implying that it can be easily and effectively applied to procedural and object-oriented paradigms, etc.

One of the characteristics of Cleanroom software development that seems to have caused controversy is the fact that unit testing and error correction (i.e. debugging) is substituted by correctness verification and statistical testing. While Cleanroom proponents take pride in this fact due to the realised development cost savings (Trammell et al., 1992), others consider it a dangerous malpractice which contradicts "known testing theory as well as common sense" (Beizer, 1997, p. 14). Beizer (1997) contests the effectiveness of Cleanroom by arguing that "Its claimed superiority is based on grievously flawed attempts at controlled experiments." (p. 14), implying that the appealing results reported in literature are based on unfair and sometimes invalid comparisons; Beizer (1997) notes further that:

"When advocates "compare" Cleanroom with testing, it is always with some obsolete hacking model of a quarter-century ago. Cleanroom is never measured against

- proper unit testing done under coverage standards,
- people trained in testing techniques,
- shops that use test design and automation technology, or
In addition, Beizer (1997) criticises the excessive reliance of Cleanroom on statistical testing and suggests that, while Cleanroom promotes the statistical testing of high probability operational profiles, the most serious errors may be occurring in low probability operational profiles. Unfortunately, there is no evidence that such profiles are adequately tested for errors. Furthermore, while Cleanroom proponents claim that correctness verification is powerful for avoiding error injection, as mentioned above, correctness verification is not always applied in practice in the way suggested by the theory. This is due to cost considerations. Furthermore, while Cleanroom software development is claimed to result in a near zero error level and to produce better code than unit testing, it also specifically requires the involvement of highly skilled and committed developers (Pressman, 1997).

Although success stories are reported in the literature, it is not known how effective the Cleanroom model can be when applied by less-skilled and less-committed developers (Linger, 1994; Sommerville, 1996; Trammell et al., 1992). In addition to Beizer's (1997) critique, this is probably one of the reasons why, despite a quarter of a century of publicity, Cleanroom has not become part of mainstream software development (Beizer, 1997).

2.3.4 Preventing Errors using Metrics
In this section, the use of metrics to prevent errors is discussed. Unlike the studies reported in section 2.2.3, which measure various artifact properties to help detect error-prone sections in an artifact, error prevention studies use error counts as metrics to analyse error injection and removal patterns in an effort to prevent such errors in future development exercises. Section 2.3.4.1 focuses on the Personal Software Process, whereas section 2.3.4.2 discusses phase containment metrics.
2.3.4.1 The Personal Software Process (PSP)

In his book entitled “A Discipline for Software Engineering”, Watts Humphrey (1995), describes a further error prevention methodology, which has been developed at the Software Engineering Institute. The approach is known as the Personal Software Process (PSP) and aims to assist software developers in learning about their personal error propensities over different software development phases and utilises such knowledge to prevent errors. Developers use PSP by collecting data on the basic metrics of development time spent on a phase and errors committed, detected, and corrected. This information is recorded in Time and Error Recording Logs. The process of personal data collection and recording is repeated over a number of projects and the recorded information is retained in project summaries and analysed with the aim of discerning personal trends of error injection and removal. Information on the likely distribution of error types over various development phases, and on likely time intervals required to locate and fix errors etc. is expected to emerge during the analysis of the collected data. The information on personal trends is expected to make developers aware of what their strengths and weaknesses are and to improve the way that developers build software so that error introduction can be averted. Admittedly, there is limited experience with the application of PSP. PSP trials at six universities and in three software organisations have consistently demonstrated that the use of a defined and measured PSP has improved the quality (i.e. terms of fewer errors committed) of development artifacts five to ten times, while also improving developers’ productivity (Ferguson, Humphrey, Khajenoori, Macke, & Matvya, 1997; Hayes & Over, 1997; Humphrey, 1994a, 1994b, 1995, 1996; Silderberg, 1998). While the limited evidence is encouraging, the cost incurred by the developers to become PSP-proficient is only alluded to. PSP data are required to be collected over many projects before personal trends can emerge. In addition, Disney & Johnson (1998) in their PSP study observed that participants complained about the number of data recording activities, which were regarded time-consuming and disruptive to their normal flow of work (Disney & Johnson, 1998).

While Disney & Johnson (1998) support existing PSP results, they question whether PSP results are highly dependent on the quality of data collection and analysis. According to their experiment results, three types of anomalies, namely, omissions, additions, and transcriptions, may affect PSP data collection. Omission anomalies occur
when a developer fails to record required measures of errors committed, time etc. Addition anomalies occur when the developer unintentionally makes up new data that does not necessarily reflect the actual practice. Transcription anomalies occur when the developer makes a mistake in the process of data recording. Similar anomalies may occur even during the PSP data analysis. For example omission anomalies may occur when the developer fails to perform a required analysis on the collected data. Calculation anomalies take place when incorrect calculations are performed and transcription anomalies occur during data transfer.

The possibility of such anomalies during the data collection and analysis suggests that PSP data alone may not be used to assess the effectiveness of the PSP methodology due to the possibility of such data being erroneous. Disney & Johnson (1998) illustrate this point as follows:

"The improvement in average defect [error] levels for engineers who complete the course is 58.0%, if based upon PSP data alone, might only reflect a decreasing trend in defect [error] recording, not a decreased trend in the defects [errors] present in the work product" (p. 151, 1998).

Another drawback relates to the application of PSP in software development organisations where management support has been critical on its success. In this context McAndrews (2000) observes:

"... It turned out that the PSP skills were not effectively practised within organisations unless those organisations made a commitment to the PSP approach and to the individuals who used it. Humphrey found that if managers do not provide a supportive environment, and ask for and constructively using PSP data, engineers soon stop using PSP. In these cases, engineers fell back to the chaotic practices that they had used before they were PSP-trained." (p. 3)

As a consequence, recently, Watts Humphrey extended PSP to develop the Team Software Process (TSP) (McAndrews, 2000). TSP provides guidance to teams of PSP-trained developers on how to articulate the artifact quality (in terms of lack of errors), productivity and schedule goals with which managers can identify. Admittedly, this has helped developers gain management support for their activities. TSP is also said to greatly enhance team dynamics and continuously improve software development to
avoid errors. Only few empirical results have been published, yet the results produced are compelling. Different studies are consistent in their reports about significant reductions in densities of errors, test duration, early error detection and removal while using TSP (McAndrews, 2000; Webb, 2000; Webb & Humphrey, 1999).

Despite any of the PSP pitfalls discussed above, it is widely agreed upon that PSP has a significant educational value with a considerable positive impact on software engineers who have adopted it in their development practices.

2.3.4.2 Hevner's Phase-Containment Metrics
Hevner (1997) discusses another approach to error prevention that is based on the notion of the phase containment of errors. The phase containment of errors means that any error that enters a software artifact should be located and corrected within the same development phase in which it was injected (Hevner, 1997). In this context, in-phase and out-of-phase errors are defined. An in-phase error is an error that is found in the same phase in which it originated. If an error escapes the phase in which it was introduced and is located and corrected in subsequent development phases, it is then referred to as an out-of-phase error. Data consisting of counts of in-phase and out-of-phase errors are collected during the various phases of development of selected software projects using formal inspection techniques. The analysis of the collected data on in- and out-of-phase error counts produces metrics showing total errors by phase and total errors that escape the phase in which they were injected. These metrics (also known as phase containment metrics) are expected to help developers gain critical insights on the existing software development practices and their own error-prevention ability. Such insights are used to improve development practices for future projects. While experience on the use of phase containment metrics in the industry has been reported by Hevner (1997), this study does not appear to have been replicated.
2.3.4.3 Summary and Evaluation of Error Prevention Using Metrics

While the rationale of Humphrey's (1994) and Hevner's (1997) approaches is similar (i.e. learning from past experience by counting and analyzing one's own errors), the former targets individual developers, whereas the latter has been used with teams of developers. The other difference is that Hevner's data collection is more detailed, because it distinguishes between in-phase and out-of-phase errors. This may make Hevner's data much more insightful than Humphrey's and more effective in preventing errors. This assertion, however, requires empirical validation as it does not appear to have been addressed in the existing literature.

2.3.5 Summary and Evaluation of Error Prevention Approaches

In section 2.3, error prevention approaches were surveyed. In section 2.3.1 various ways to prevent errors in requirements artifacts were discussed. The incorporation of error cause information into error prevention was covered in section 2.3.2. In section 2.3.3, the Cleanroom software development methodology was explained. Finally, in section 2.3.4, the use of error count metrics to prevent errors was discussed.

In general, preventive approaches rely on a set of steps that must be systematically carried out in order to prevent errors. These steps do not seem to be applied easily to individuals, but have been adopted by large organisations, or experienced teams, sometimes requiring extensive mathematical skills (e.g. error prevention using cause analysis and Cleanroom software development). No evidence was found whether such steps can be easily adopted by average individual developers. One exception, however, is Yu's (1998) study where the catalogue of C coding errors was used by individual developers. Despite the fact that the prevention capability of the catalogue was limited to coding, the idea may be useful for other types of artifacts and this will be investigated further in this thesis. The approaches to software prevention using metrics, although reported to be effective, require the systematic accumulation of past software development data, which is costly and time consuming. The approaches preventing errors from requirements artifacts, although effective, have limited applicability. They should, however, be incorporated into all software development phases. In general, it was observed that the effectiveness of error prevention approaches depends on error detection approaches which indicates that the two are closely related.
2.4 Chapter Summary and Discussion

The objective of this chapter was to examine the existing literature for different approaches used to target errors in software development. The above examination suggests that two main approaches can be used to target errors in software development: the error detection approach, by which software artifacts are first developed and then systematically searched for errors, and the error prevention approach where attempts are made to avert the injection of errors into software artifacts while they are being developed. The literature shows that a significant amount of work has been carried out on error detection and, while, except for formal methods, the amount of work on error prevention is clearly less, it has received significant attention during the last decade.

While the error detection and prevention approaches are essentially different, they are strongly related to each other. This relationship is based on the fact that error detection generates knowledge about errors which presents opportunities for error prevention. Conversely, error prevention approaches make use of error detection while development artifacts are being developed. This ensures that the detection of injected errors occurs as close as possible to the point at which they originate. This relationship is summarised in figure 2.6.

![Figure 2.6 - Relationship between Error Detection and Error Prevention](image)

This pattern can be seen in many error prevention studies (Kajijara et al., 1993; Mays et al., 1990; Yu, 1998) etc. Jorgensen (2002) argues that no matter how successful an error detection approach is, it always suffers from two problems:

i) a developer cannot know the total number of errors in an artifact, and

ii) no matter how many errors are detected in an artifact, a developer can never prove that the artifact is error free.
Therefore, in error detection it is better to work backward from errors. That is, given particular errors, a developer can choose error detection approaches that are likely to uncover such errors. This may constitute a pragmatic approach to enhance error detection effectiveness. This view is also supported in Ciolkowski, (1999); Jorgensen, (2002); Lanubile et al., (1998). Consequently, knowledge about errors is important. Overbeck (1994) and Andersen (1996) also suggest that knowledge of errors can help in the design of error detection approaches (Andersen, 1996; Binder, 2000; Overbeck, 1994), which means that, the more we know about errors, the better the error detection approach. It can, therefore, be concluded that error detection works best if developers know about the errors that they want to detect. However, this approach may not be easy in practice, because software in different application domains may contain errors of different types. For example, errors in business transactions processing software may be quite different from errors in embedded software (Pressman, 1997; Sommerville, 1996). The examination of literature did not find any publication that categorises error types by application domain. This suggests that there are no systematic guidelines for developers about what error detection approaches to use in order to detect errors in a given application domain. Consequently, developers are likely to make arbitrary decisions about what error detection approaches to use or they are likely to make decisions that are based on prior experience or simply based on hunch and gut feeling.

Also, even if it is assumed that such guidelines exist, the review of error detection approaches in section 2.2, has shown that there is no error detection approach that can detect all errors. In practice, developers may need to use a combination of different approaches. For example, reviews or inspections are good for detecting errors in early software development artifacts (Doolan, 1992), metrics are good for detecting errors in design artifacts (Briand et al., 2000; Briand et al., 1999b), testing techniques are good for detecting errors in code artifacts (Binder, 2000). However, if applied together, various error detection approaches may increase software development costs. Also, as argued in section 2.2.5, error detection is not the end of the story, but must be followed by error correction, which is also associated with additional costs. There is evidence in literature that not all software development organisations are willing or able to incur such costs, and frequently avoid many error detection approaches altogether (e.g. reviews, inspections, etc.) (Slaughter, Harter, & Krishman, 1998) and frequently, software development artifacts or entire software products are delivered with errors.
which may cause failures during software operation. In conclusion, error detection approaches work best if developers know about what errors they are looking for, but they may be quite costly.

The relationship between knowledge of errors and error prevention is important, because it suggests that error prevention can occur if developers know what errors should not be injected into software development artifacts. Slaughter, Harter, & Krishnan (1998) argue that error prevention is superior to error detection when cost considerations are made. They suggest that in order to prevent errors from being introduced into software artifacts, additional work must be carried out during artifact development which may include the early use of error detection approaches. The additional error prevention work incurs additional cost, which according to Slaughter, Harter, & Krishnan is voluntary, because developers undertake error prevention work voluntarily. On the other hand, the cost that would be incurred by developers to correct errors after detection, when error prevention work is not carried out, is involuntary. This is because developers incur such costs due to their contractual obligations towards their clients, not because they are willing to return to and fix software that they have already delivered. According to Slaughter, Harter, & Krishnan (1998), any increase on voluntary error prevention cost is offset by the savings gains due to the decreased involuntary costs. This view is strongly supported by Crosby, (1979); Hendrickson, (2001); Juran & Gryna, (1988); Thielen, (1992). Ruthven (2002) cites an example from IBM where error prevention voluntary costs incurred were approximately at $90/error as opposed to field involuntary corrections costs, which amounted to $25,000/error. In this example error prevention is more than 250 times cheaper than error detection and correction (Ruthven, 2002).

It would be a mistake, however, to totally dismiss error detection, as the relationship between error prevention and error detection is strong as explained by Henrickson (2001):

"The key to improving the quality [in terms of fewer errors committed] of your software is to invest heavily in bug [error] prevention and very early defect [error] detection. It's best if the bug [error] never happens. But if it does, the closer to the original developer's desk the bug is found, the better." (p. 10)
This means that error detection approaches not only help error prevention approaches by building knowledge about errors, but also by detecting errors early (Hendrickson, 2001).

In summary, error prevention is better than error detection, but if error detection is applied early it can consolidate error prevention. Given that the objective of this research is to construct an error prevention approach (see chapter one), it is worthwhile to briefly review the salient features of the error prevention approaches that were examined in section 2.3. Most error prevention approaches using error causal analysis and Cleanroom software development require an organisational infrastructure in place (e.g. repositories for error information storage, error information dissemination systems, etc.), are team based (e.g. cause analysis meetings), and require experience and strong mathematical skills (e.g. Cleanroom software development, mathematical correctness verification skills). Error prevention approaches using metrics (both PSP and phase-containment metrics) require significant data collection from prior software projects in order to be useful. The error prevention approaches for requirements artifacts that are useful, are also limited in their scope. When viewed in the context of the other error prevention approaches, one approach that was found successful, without being significantly costly, was described by Yu (1998) with the catalogue of C errors. Yu’s idea is worth investigating further for four main reasons:

i) A catalogue of errors to help error prevention is not as costly as the other error prevention approaches.

ii) A catalogue of errors to help error prevention has the potential to be comprehensive for all phases of software development.

iii) In order to build a catalogue of errors, information about errors would need to be studied first. Answers to the following questions would not only be beneficial to error prevention but also to error detection (see figure 2.6 and refer to (Lanubile et al., 1998):

a) How much is known about errors?

b) What is known about errors?

c) To what extent is what is known about errors important?

d) Is knowledge about errors systematically organised?

iv) A catalogue of errors may serve as a training and educational tool. This is important because as will be seen later the lack of developer education and training constitutes one of the major causes of software errors (Yu, 1998).
These issues, the benefits arising from addressing the above issues and the ways to address them are formalised as a research thesis in the next chapter.
3. Research Questions, Significance, and Approach

3.1 Overview

This chapter has three major objectives. First, it explains the research questions that this thesis has pursued. Second, it outlines the significance of each research question. Third, the factors determining the success of the investigation of each research question are addressed. Fourth, the chapter concludes with the presentation of a hybrid research approach consisting of two field studies and a field experiment which were used to address each research question.

3.2 Research Objective and Questions

As chapter two has shown, there are two possible approaches to target errors, namely, the preventive and the detective approach. The preventive approach is favoured against the detective approach because it not only attempts to avert errors from being injected into development artifacts from the start of the development, but it is also much cheaper than its detective counterpart. Despite this fact, the detective approach cannot be totally ignored because it has helped build up considerable knowledge about errors. In addition, the detective approach can help developers gain the confidence that constructed software artifacts do not contain errors early in the development.

In chapter two, it was also pointed out that most error preventive approaches have drawbacks. For example, some error preventive approaches are difficult to apply by the individual developer, whereas others are limited to a single development phase, and yet others require excessive amounts of data collection and experience. The study of Yu (1998), however, which presented the development of a catalogue of C code errors and its use in the development of C programs, while limited to code artifacts, was found to be an interesting avenue for further research.

The overall objective of this research has been to construct an error preventive approach using a catalogue of errors that encompasses the entire development (i.e. all development phases). The catalogue of errors can be used to prevent errors from being introduced into the common software development artifacts. In order to achieve this objective, a number of issues must be investigated. First, the catalogue of errors must be
developed. Second, the catalogue of errors must be empirically evaluated in order to see whether it can indeed help prevent errors. In order to simplify the accomplishment of the overall research objective, three complementary research questions have been identified and explained in the following sections.

3.2.1 Question One: The Error Framework

_What type of error information is important to help developers learn about errors and how can such information be organised into a generic Error Framework?_

This question consists of two parts. Firstly, it seeks to investigate what information about errors is important to help developers learn about errors. There are a number of studies that have focused on errors (Andersen, 1996; Beizer, 1990; Bytesmiths, 1995; Grady, 1992; Hayes, 1994; Yu, 1998; Yu, Barshefsky, & Huang, 1997). These and others appear to be inconsistent in what error information is important. Some suggest that an error name is sufficient (Bytesmiths, 1995), some present taxonomies of errors, where they organise errors in different categories based on different criteria (Beizer, 1990), others suggest that error descriptions including examples are required to enhance developers' understanding of errors (Andersen, 1996), while yet in others, the focus shifts to error causes and prevention countermeasures (Yu, 1998; Yu et al., 1997).

Therefore, this thesis reports on a comprehensive investigation of errors. This investigation has been made from different perspectives. The identification of important error perspectives can assist the understanding of and learning about errors, if such information can be presented in a meaningful generic way, which dictates what needs to be known about errors. The second part of the question will investigate how the identification of error perspectives helps our understanding of them. Efforts to determine what needs to be known about errors have been made in existing literature. Examples include the efforts made in Freimut, (2001); Purchase & Winder, (1991). The results of these efforts, however, do not seem to be comprehensive and are quite inconsistent with each other.

Given that the overall objective of the research is to construct a catalogue of errors to help in error prevention, it is clear that prerequisite efforts must be undertaken to determine in what way information about errors must be built up. Consequently, a
generic error template or framework is needed. The Cambridge International Dictionary of English defines the term framework as: "a supporting structure around which something can be built".

For the purpose of this research question, an Error Framework is defined to be a structure that consists of a generic collection of perspectives from which errors must be analysed. The analysis of errors from this collection of perspectives is expected to generate information that needs to be known about an error, if its prevention is to be facilitated.

In summary, research question one constitutes an attempt to identify what perspectives about errors are important in enhancing developer understanding and learning, in order to facilitate their prevention. Also, once error perspectives are identified, research question one should also address the issue of how the identified error perspectives can be organised into a generic Error Framework.

3.2.2 Question Two: The Catalogue of Errors

How can the Error Framework (developed in question one) be used to catalogue errors that are commonly injected into various software development artifacts?

A Catalogue of Errors is defined as an organised collection of software development errors, where individual errors are described with specific information concerning all of the perspectives that constitute the Error Framework.

The idea of a catalogue of errors is not new. Several studies have attempted to develop catalogues of errors (Lazonder & Van Der Meij, 1995; Yu, 1998) etc., however, they have deficiencies. First, they are not comprehensive, because they focus on a single development phase or artifact (i.e. requirements, design, or code). For example, requirements errors have been catalogued in Lutz, (1993), design errors have been catalogued in McGregor & Sykes, (1992), and code errors have been catalogued in Andersen, (1996); Belzer, (1990); and Yu, (1998) etc. Some might argue that this deficiency may be easily overcome by simply compiling the different catalogues. While this option cannot be ruled out, the constituent parts of a resulting catalogue of errors
would be largely inconsistent with each other, because different studies catalogue errors in different ways. In this study, errors from the various development phases or artifacts would be catalogued on the basis of a single, unique Error Framework to generate a consistent Catalogue of Errors that encompasses all phases of software development.

In order to fully address this question a number of corollary questions need to be tackled, for instance, what errors in what artifacts will constitute the focus of the Catalogue of Errors. If the Catalogue of Errors encompasses the entire software development (including the coding phase), which programming language will be focused on. Due to the time constraints of this study, some parts of the Catalogue (e.g. code errors) will be programming language dependent, while others (e.g. design errors) may be development paradigm dependent. This is because a selected programming language would belong to a specific paradigm. Issues of this nature are expected to cause limitations in the proposed Catalogue of Errors, because it would not be feasible to catalogue each error in every language in every paradigm.

### 3.2.3 Question Three: The Impact of the Catalogue of Errors on Software Development

**What is the impact of using the Catalogue of Errors on software development?**

The aim of this question is to investigate whether the Catalogue of Errors helps developers prevent errors. There are several preventive approaches, but some are expensive and time-consuming to apply. Some even require considerable experience and expertise. A Catalogue of Errors, while requiring time for developers to master, will help them not to commit the errors that they have learned about in the Catalogue, in the first instance. In other words, using a Catalogue of Errors may be a much cheaper alternative in preventing errors than other preventive approaches. Hence question three in more concrete terms generates the following two subquestions:

1. **Does training software developers with the Catalogue of Errors (developed by addressing question two) help reduce the number of errors injected by them into software development artifacts?** If yes, can the reduction of the injected
errors be quantified? “Reducing the number of errors” is defined to mean the following:
a) preventing errors from being introduced into an artifact; or
b) identifying and correcting errors injected in an artifact before the construction of subsequent artifacts starts.
ii) What is the effect of the use of the Catalogue of Errors on the productivity of software developers? Can this effect be quantified?

3.2.4 Summary: Relationships between Questions
In the above sections, the questions to be addressed in this thesis have been defined. Clearly, the questions are strongly related to each other, in the sense that the results produced by addressing one question are used as a basis to investigate the subsequent question. This is an indication that, taken together, the questions of this research are cohesive and serve a single objective: the construction of an error preventive approach using a Catalogue of Errors. The relationship between the questions has been summarised in figure 3.1.

Figure 3.1 – Relationships between Research Question 1, 2, and 3
Figure 3.1 shows that the Catalogue of Errors is divided into three parts. Figure 3.1 also shows that only three development phases and their respective artifacts are addressed. The rationale for these decisions is presented in chapter seven, where the development of the Catalogue of Errors is addressed in more detail.

3.3 Motivation

The objective of this section is to present the motivation for the potential benefits that are expected to arise by addressing each research question (see section 3.2). The decision to address the motivation of each question separately rather than collectively was deliberate. This was done in order to show that while the three questions are cohesive and serve a unique research objective (i.e. develop an error prevention approach using a Catalogue of Errors) the outcomes expected to be obtained after addressing the individual questions would be beneficial, even if they were to be considered in isolation.

3.3.1 Question One: Why develop an Error Framework?

There are several reasons why an Error Framework needs to be developed. Firstly, the Error Framework is expected to provide a solid basis for identifying and supporting major issues with respect to error handling. Error handling issues are comprised of, but are not limited to issues such as error description, error identification, error cause analysis and prevention etc. (Humphrey, 1989b; Lezonder & Van Der Meij, 1995). Secondly, the different perspectives of the Error Framework may help characterise the quality of the processes and activities that are carried out to construct an artifact (Basili & Weiss, 1984; Freimut, 2001; Hendrickson, 2001). Thirdly, the Error Framework is expected to provide a template dictating the accumulation of consistent information about errors (Florac, 1992). The template provided by the Error Framework assures that important questions about errors are answered. Brown et al. (1998) describe various software architecture patterns that can cause problems. Such patterns are also known as antipatterns. Brown et al. (1998) propose and use an antipattern template to consistently describe their antipatterns and argue the need for it, by saying:

"Each [anti]pattern has a consistent, rhetorical structure-a patterns template. By rhetorical structure, we mean that there is a well-conceived logic to the pattern descriptions. The consistent logical structure is a direct result of the use of the
In this context, the proposed Error Framework is expected to serve the same purpose as Brown et al.'s (1998) anti-pattern template.

Lastly, an Error Framework can be used by developers as a template to create mental representations of errors and, therefore, to help developers organise and associate knowledge structures about errors accordingly (Borgman, 1999; Stone, Jordan, & Wright, 1990). This is particularly important for student or novice developers who seem to lack the ability to develop associations among the different aspects of errors (e.g. association between errors, their symptoms and their causes) (Allwood & Bjorhag, 1990).

3.3.2 Question Two: Why develop a Catalogue of Errors?

The development of a Catalogue of Errors is important for several reasons. Firstly, a Catalogue of Errors can be used as an educational guide for software development practitioners and students (Fraser, Smith, & Smith, 1992). The education of developers is important because the quality of a development artifact (in terms of lack of errors) is governed directly by the individual who is supposed to develop it and his/her knowledge (Slaughter et al., 1998). A Catalogue of Errors, in its capacity as an educational guide, can work on several dimensions including the ability to (Brown, Malveau, McCormick III, & Mowbray, 1998; Cooper, 2000; Gamma, Helm, Johnson, & Vlissides, 1995; Purchase & Winder, 1991):

i) Establish awareness about common errors;
ii) Identify common errors;
iii) Establish solutions about common errors;
iv) Avoid common errors;

Such dimensions can collectively help reduce the risk that developers will commit such errors in their future development efforts (Yu, 1998). As a corollary argument to this point, it can be added that knowledge of errors in a Catalogue may also help enhance
the error correction (i.e. debugging) skills of developers, while minimising the number of new errors that may be introduced during the correction of old ones (Stone et al., 1990). Secondly, a Catalogue of Errors is meant to be free from application domain dependent details. This implies that errors in the Catalogue are described generically and can, therefore, be of value to any developer irrespective of their degree of application domain expertise (Lanubile et al., 1998; Mays et al., 1990).

Thirdly, a Catalogue of Errors is meant to document typical errors, and knowledge of such errors may also help software engineers know more about the quality of the processes that they carry out to produce software artifacts. In this context, Basili & Weiss (1984) state:

"To obtain insight into the software development process, the data collectors need to know the kinds of errors committed ... ." (p. 729).

Lastly, while the objective of this Catalogue of Errors is to help prevent the errors it documents, it can also be used by developers who believe in error detection rather than error prevention (Paulk, Weber, Curtis, & Chrissis, 1994). In this context, Travassos et al. (1999a) refer to the detection of object-oriented errors by saying:

"Before we can describe how to detect OO defects [errors], we need to have some knowledge of the different kinds of defects [errors] to be sought." (Travassos et al., 1999a).

3.3.3 Question Three: Why Investigate the Impact of a Catalogue of Errors?
The investigation of the impact of a Catalogue of Errors on software development is important for a number of reasons. Firstly, this question is an attempt to investigate whether a Catalogue of Errors can actually help increase awareness about errors, to help developers quickly detect their presence in an artifact, and resolve and even avoid them. Therefore, this question can help validate the work done to answer question one and two. Secondly, the investigation of this question is important because the results may help restructure the way developers approach the construction of software artifacts (i.e. using a Catalogue of Errors to identify potential development pitfalls before embarking on the construction of the artifact) (Tichy, 1998). Thirdly, there have been other similar studies that have indicated that a catalogues of errors helps prevent such errors. As shown in Yu (1998), a catalogue of C code errors was found to be successful. This
study, however, focused only on C code errors, ignoring pre-code errors. Existing research shows that most errors introduced into software artifacts are pre-code errors (Boehm & Basili, 2001; Duncan et al., 1999; McGregor & Korson, 1994a; Roper, 1994). This suggests that including pre-code errors in the Catalogue and testing its impact of on the development of pre-code artifacts, might be worthwhile. Fourthly, there is ample evidence in the literature stressing the importance of empirical validation in computer science and software engineering (Mitchell & Welty, 1988; Tichy, 1998; Tichy, Lukowicz, Prechelt, & Heinz, 1995). While some may find the answer to question three intuitive and maybe predictable, this intuition should only be used as a starting point and must be backed by empirical evidence (Tichy, 1998). In this context, Lewis, Henry, & Kafura (1992) argue that sometimes software engineering literature is littered with unsubstantiated claims, for example,

"... claims made by software engineers often remain unsubstantiated because they are inherently difficult to validate or because their intuitive appeal seems to dismiss the need for scientific confirmation." (p. 173) (Lewis, Henry, & Kafura, 1992).

This statement suggests that the answer to any question, irrespective of its predictability, must be empirically validated, no matter how intuitive it may seem.

3.4 Success Factors for the Research Questions
The objective of this section is to define factors that allow the investigator to determine whether a given research question has been successfully addressed. This is important because it not only helps to understand the research questions better, but also and most importantly it provides the necessary ingredients to address each research question. In this section, the success factors are clearly identified and the rationale underlying their importance is explained. The success factors are categorised on the basis of the research questions and they are addressed in that order.

3.4.1 Question One Success Factors
Question one requires the development of an Error Framework which covers various perspectives from which errors can be analysed. Purchase & Winder (1991) and Freimut (2001) suggest that the perspectives of any error framework should be orthogonal,
comprehensive and useable. The orthogonality of the perspectives of any error framework is important because the perspectives should be mutually independent in order to ensure that every piece of information about an error is considered by a single perspective only. The perspectives of any error framework should be as comprehensive as possible in order to ensure that no information about errors that can help in error prevention or early detection is left unaccounted for. The perspectives of any error framework should be useable so that the error information that they represent can be easily derived and used to build up the Catalogue of Errors.

As suggested by section 3.2, the Error Framework is meant to be used as a template to build a Catalogue of Errors which would then be used by developers. Consequently, it is crucial that the perspectives that constitute the Error Framework be empirically evaluated by the developers in terms of their usefulness. The usefulness of an error perspective constitutes the practical value of the perspective as perceived by the developer. The determination of the usefulness of the perspectives of the Error Framework is important because if developers regard a given perspective as being not useful, they are probably not going to use the information that can be derived by analysing errors from that perspective. Therefore, usefulness should be added to the list of the success factors that is suggested by Purchase & Winder (1991) and Freimut (2001).

In summary, the factors that can be used to assess the success of the outcome of question one, i.e. the Error Framework, are orthogonality, comprehensiveness, useability, and usefulness. These factors have been systematically addressed in chapter six, which is where the Error Framework (i.e. research question one outcome) is proposed.

3.4.2 Question Two Success Factors

Question two requires the development of a Catalogue of Errors by using the Error Framework constructed in question one. The objective of the Catalogue of Errors is not to simply act as a repository of errors. It should properly implement the perspectives of the Error Framework. The Catalogue of Errors should also be easy for developers to learn and easy for them to use and be helpful in addressing the errors that it describes during the development of various software artifacts. Finally, the knowledge about
errors that is disseminated by the Catalogue of Errors should have a positive impact on developers' overall software development skills. Therefore, the success factors for the outcome the Catalogue of Errors include the following:

i) Learnability;
ii) Implementation of Error Framework perspectives;
iii) Useability;
iv) Usefulness in identify and correct errors; and
v) Overall Software Development Skills Improvement.

3.4.3 Question Three Success Factors

Question three requires the assessment of the impact that the Catalogue of Errors has on the ability of developers:

i) to prevent errors from being introduced in an artifact,
ii) to identify and correct errors injected into an artifact before the construction of the subsequent artifacts starts, and
iii) to be productive in the development of software artifacts.

Clearly, a positive impact of the Catalogue of Errors on all the elements of the above list would constitute a successful outcome for question three. The elements of the above list are therefore, considered as the success factors of research question three. However, in order to determine the outcome of question three (i.e. the impact of the Catalogue of Errors) with regard to the above success factors, a number of measurements must be considered and made. For example, the impact of the Catalogue of Errors on the ability of a particular developer to prevent errors cannot be determined unless the actual errors that this developer commits are counted and their type determined. Similarly, the impact of the Catalogue of Errors on developer productivity cannot be assessed unless the proportion of a development artifact that is constructed in a given amount of time is known.

The complete list of the measurements that can help determine the outcome of question three and the issues that are related to them have been explained in the following sections. These measurements are referred to again and addressed in chapter eight, which is where the outcome of research question three (i.e. impact of Catalogue of Errors) is examined.
3.4.3.1 Error Counts and Types

The justification for using error counts and error types as success factors is self-evident. As question three suggests, the principal goal for the development of the Catalogue of Errors is to reduce the number of errors that developers commit in the various development artifacts. Consequently, it would be natural to keep an account of the number and type of errors committed by developers who use and do not use the Catalogue of Errors and compare error counts to get some insight as to whether the Catalogue of Errors actually helps developers to commit less errors.

Error counts are used in many studies (Conte et al., 1986). Early examples include Litecky & Davis (1976) who counted errors in Cobol to find out those parts of the language that were more prone to errors (Litecky & Davis, 1976). Later, Sunsoft, a commercial software development company, has used error counts to measure the quality of its Infinite Plus software (Kitchenham, 1996). Another more recent example where error counts were used to demonstrate the effectiveness of reading techniques to discover errors in requirements artifacts, is the study of Shull et al. (1999) who "collected quantitative data by having the subjects turn in their defect [error] lists." (p.7). Many other similar studies have obtained a simple error count, but have ignored the types of errors committed (Conte et al., 1986; Wohlin, 1998; Wohlin et al., 2000).

If error counts were to be used alone, they would not be completely helpful to address question three. The expectation is that when developers have access to the Catalogue of Errors (which contains limited types of errors\(^\text{12}\)), they will commit fewer catalogued errors, because they would know about them. Therefore, error type evidence should be collected as well, that is error counts should be organised by error type.

Nevertheless, the use of error type count information to determine what and how many errors are injected into artifacts suffers from one potential problem. If a developer produces only part of an artifact, then the determination of error type count has the potential to be misleading. In circumstances where only part of an artifact is examined, it is sensible to assume that fewer errors are likely to be found. This, however, would

---

\(^{12}\) This decision was made because of the time limitation constraint. A more detailed discussion is made in chapter 7.
not be an indication that a developer is better because he/she has committed fewer errors. To address a situation like this, it was decided that error type counts need to be normalised by dividing them by the size of the artifact where they are injected (Kitchenham, 1996; Land, Jeffery, & Sauer, 1997). Kitchenham (1996) explains the term ‘normalise’ by saying that:

“In order to compare the quality of different software products, defect [error] counts should be ‘normalised’ by dividing [error counts] by the product size…”
(p.83).

Thus, normalising error type counts was achieved by dividing the error type counts by the size of the artifact. This is known as the error type density of the artifact or simply error density (Kitchenham, 1996).

3.4.3.1 Error Density

Card & Glass (1990) define the ratio between the number of errors and the size of the artifact as the error rate. Fenton & Pfleeger (1997) define the same ratio as error density. It was decided that for this study the term ‘error rate’ was inappropriate and misleading because it implies some suggestion of time, which error density does not (Fenton & Pfleeger, 1997). Kitchenham (1996) supports the use of error density:

“…the size of the software product is an element which influences all dimensions [e.g. quality, productivity, etc.]. It allows us to ‘normalise’ measures in each dimension to enable cross-project comparisons. In this context the term ‘normalise’ means converting a measure into a rate or percentage.” (p. 64)

This suggests that error density is useful to compare development artifacts when they are developed using different techniques or by different authors.

Conte, Dunsmore & Shen (1986), however, suggest that error density is flawed:

“Our analysis leads to the conclusion that error density is generally a poor size-normalised index of program quality.” (p.349).

They base this conclusion on the findings of some studies (Motley & Brooks, 1977; Potier, Albin, Ferreol, & Bilodeau, 1982; Shen, Yu, Thebaut, & Paulsen, 1985). These studies have observed that a program or an artifact size is not proportional to the
number of errors in it. This finding is unexpected and contrary to the general belief that larger programs are susceptible to a larger number of errors (Conte et al., 1986).

Also, Conte, Dunsmore, & Shen (1986) conclude that the minimum program size beyond which the error density could reasonably be considered unrelated to size is approximately 500 lines of code. This conclusion may not be entirely accurate, because there is no mention about how complex 500 lines of code should be, in order for program size to be proportional to error density. Complexity directly affects the number of errors in the program (McCabe, Dreyer, Dunn, & Watson, 1994). No mention is made either of the relationship between the size of pre-code development artifacts and the number of errors in these. Despite this conclusion, by Conte, Dunsmore, & Shen (1986), many studies have used error density without discriminating between smaller or larger programs, for example, Bell & Thayer, (1976); Card & Glass, (1990); Daskalantonakis, (1992); Diaz & Sligo, (1997); Fenton & Pfleeger, (1997); Humphrey, (1996); McGregor, (1998a); Tsuda, Morioka, Takadachi, & Takahashi, (1992); and Wohlwend & Rosenbaum, (1994). This suggests that despite the flaw reported by Conte, Dunsmore, & Shen (1986), error density has been widely used and was, therefore, used in this research.

An error density measurement can be helpful to address question three because its values can be interpreted as follows: A low error density for any artifact means that fewer errors are injected into an artifact or that more errors are prevented from being injected into an artifact. A high error density for any artifact means that more errors are injected into the artifact or that fewer errors are prevented from being injected into an artifact. However, while error density is more accurate to compare artifacts, it also increases cost, because it requires an additional factor to be considered, namely, the size of the artifact for which error density is computed. The size of development artifacts is considered in more detail in the following section.

3.4.3.3 Size of Artifact
The determination of the size of a development artifact depends on the type of artifact. Two types of artifacts are recognised:
i) Those artifacts which are mainly textual in nature, for example, requirements artifacts describing the requirements of a system and design artifacts describing the design of the system.

ii) Those artifacts which contain lines of code in a given programming language, for example code artifacts contain lines of Java, C, C++ etc..

The literature suggests that the size of different types of artifacts is determined in different ways. For example, the size of artifacts that are textual in nature is determined by counting the pages of the document where the artifact is described. The size of artifacts containing lines of code is measured by counting the number of lines of code in the artifact. The following sections describe these in more detail.

Page Counts for Textual Artifacts

Fenton & Pfleeger (1997) argue as follows:

"... a [requirements] specification or design [artifact] can consist of both text and diagrams, suggesting at least two types of atomic objects that are incommensurate with respect to length. We can enforce comparability artificially by defining a page as an atomic object, so that both text and diagrams are composed of a number of sequential pages. Thus the number of pages measures length for arbitrary types of documentation; it is the most frequently used in industry." (p. 255).

This type of measure for textual artifacts in general and for requirements and design artifacts in particular has been used in various studies (Bassman, McGarry, & Pajerski, 1995; Bell & Thayer, 1976; Gilb & Graham, 1996; McGregor, 1998a; Peng & Wallace, 1993) and was therefore used in this research as well.

Number of Lines of Code (LOC) of Code Artifacts

As suggested above, the size of code artifacts is determined by the number of lines of code (LOC) in the artifact. It is debatable whether or not LOC may be used to determine the size of a software artifact accurately. A number of problems that are often attributed to LOC have been reproduced below from (Boehm, 1987b):
i) The LOC measurement may be problematic because complex statements or complex combinations of statements receive the same weight as simple assignment statements;

ii) The LOC measurement is not a uniform metric because lines of machine-oriented language, or higher-order language, or very high level language are given the same weight;

iii) The LOC measurement is difficult to define well, especially when deciding what should be included and/or excluded in the line count (e.g. comments, non-executable lines of code, etc.).

iv) The LOC measurement is not well correlated with the value added to software development; unnecessary statements may be added without necessarily adding to the real value.

v) The LOC measurement does not necessarily represent quality; sloppy code may be added which may not necessarily be free from errors.

The nonuniformity issue of LOC is raised in relation to languages by Lorenz & Kidd, (1994). They suggest that in different languages LOC counts are different to determine the size of code artifacts. While this may be a problem when projects developed with different programming languages are compared, it can be avoided if projects have the same language and use the same programming conventions.

Other problems with LOC are related to the definition of LOC. Definitions differ on whether they include or exclude blank lines, comments, and data declaration lines of code. Conte, Dunsmore, & Shen (1986) and Grady & Caswell (1987) define a line of code as any line of the program text except comments and blank lines. This definition includes executable and non-executable statements (e.g. declaration statements) (Conte et al., 1986). Littlefair (2001) cites other sources which exclude non-executable lines of code (p. 83). In this study, it was decided that declaration and initialisation statements cannot be exempted from the LOC count, because they can be a source of serious errors (Irvine & Offutt, 1995). Blank lines and comments, however, while they may be likely to affect the readability and maintainability of code artifacts, do not constitute a source of errors per se. Therefore, it was decided that the definition of LOC would exclude blank and comment lines.
Another issue that appears to affect LOC’s validity is that individual developers have stylistic programming differences, in the sense that some may prefer to have many statements in the same line, while others may prefer to have a single statement per line, etc. While individual stylistic programming differences can be a problem, they can be avoided if developers use the same code conventions, such as those promoted by Sun Microsystems to code Java programs etc (Sun Microsystems & Services, 1998).

Boehm (1987b) suggests that the number of LOC is not the only way to determine the size of code artifacts. Alternative ways exist. One example is using function points, however, this method is outside the scope of this chapter. The interested reader is addressed to Albrecht (1983) and Featon & Pfeeger, (1997) for a more detailed discussion about function points. Despite this, Boehm (1987b) concludes that no alternative ways have proven to be superior to LOC because some have advantages over LOC in some situations, but they also have more difficulties. Besides, the LOC is quite popular because it is (Boehm, 1987b):

i) relatively easy to define and discuss without running into ambiguities;
ii) easy and practical to measure;
iii) conceptually familiar to developers; and
iv) easily used in conjunction with other measurements (e.g. productivity)

In summary, given the advantages and disadvantages, LOC is the most practical code artifact size measurement that can be used in error density computations.

3.4.3.4 Escape Ratio
An additional factor that is related to the error type count factor is the escape ratio. The notion of escape ratios has been discussed in Havner (1997), who argues that errors in a software artifact must be identified and removed from the artifact where they have been injected, rather than from subsequent artifacts. For example, requirements artifacts are produced before design artifacts and are used as input for them. Similarly, design artifacts are produced before code artifacts and are used as input for them. Errors which are introduced in a requirements artifact and which are found before the design artifact is started are cheaper to correct than those introduced in a requirements artifact and found after the design artifact (or even the code artifact) is completed. The latter category of errors is known as escapes because they have remained undetected and
uncorrected in the artifact where they originated. The number of errors that escape the artifact where they are originated can be divided by the total number of errors introduced into this artifact (i.e. number of errors that were identified and corrected in the artifact of origin + number of escapes). This ratio is important because it indicates the ability of a developer to identify and remove errors. This escape ratio is similar to the notion of percentage escapes introduced in Hevner, (1997). A smaller value of escape ratios is better than a larger value, because it shows that fewer errors go undetected and uncorrected as development artifacts are produced. The concept of escape ratios is also known as Error Removal Efficiency and is defined in Feng & Wallace, (1993). Escape ratios can be computed for all types of artifacts and all types of errors.

3.4.3.5 Productivity
Productivity is defined as the amount of output that is produced by a developer in a unit of development time (Fenton & Pfleeger, 1997; Pfleeger, 1996; Wohlin, 1998). This productivity definition is slightly vague, because it does not specifically state what output and development time is. Commonly, output refers to the size of the artifact, whereas the unit of development time refers to the amount of time spent to develop the artifact (to be covered in section 3.4.3.6). A definition of productivity as a ratio between the size of the artifact and the time spent to develop it is widely accepted in literature (Card & Glass, 1990; Putnam & Myers, 1992; Diaz & Sligo, 1997; Tsuda et al., 1992; Wohlin, 1998; Wohlin et al., 2000; Humphrey, 1989a, 1989b, 1994a, 1994b, 1996; Humphrey, Snyder, & Willis, 1991).

The need to determine productivity as a success factor is dictated directly by question three. It is possible that using the Catalogue of Errors might slow developers down. If this is the case, productivity is directly affected. It is, therefore, important to know what effect the Catalogue of Errors has on productivity and to what extent developers themselves are affected by it.

3.4.3.6 Time Spent to Develop an Artifact
Bassman, McGarry & Pajerski (1995) suggest that the time spent to develop an artifact is measured in terms of staff hours, and is used to represent effort or cost in the development of a software artifact. Specifically they say that:
“each individual must minimally report the total number of hours of effort and a breakout of the number of hours per activity (e.g. design, code, or other).”
(p.37).

This is supported by Landis et al., (1990); Landis et al., (1992) Card & Glass, (1990); De Marco, (1979); and Kitchenham, (1996). However, while most researchers agree on the importance of the time spent to develop an artifact, caution must be taken because the measure of time spent on development can be problematic. Specifically, three important considerations need to be made when time spent is measured (Brooks, 1980):

i) time must be measured in such a way that irrelevant behaviour is excluded; and

ii) time must be measured in such a way to exclude all parts of development irrelevant to the hypothesis being tested;

Brooks’ (1980) observations are practical considerations that must be borne in mind when the time spent to develop an artifact is computed.

3.4.4 Summary: Success Factors for the Research Questions

In section 3.4 the success factors for the research question have been identified and explained. The determination of success factors is important because it allows investigators to determine whether a research question has been thoroughly investigated and whether it achieves its purpose. A summary of the success factor for each research question has been presented in Table 3.1.

Table 3.1 – Summary of Success Factors

<table>
<thead>
<tr>
<th>Research Question One</th>
<th>Success Factors</th>
<th>Research Question Three</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orthogonality</td>
<td>Learnability</td>
<td>Ability of Developers to Prevent Errors (Measurement; Error Density)</td>
</tr>
<tr>
<td>Comprehensiveness</td>
<td>Implementation of Error Framework Perspectives</td>
<td>Ability of Developers to Remove Errors in Artifacts of Origin (Measurement; Escape Ratios)</td>
</tr>
<tr>
<td>Useability</td>
<td>Useability</td>
<td>Productivity of Developers to Construct Artifacts (Measurement; Productivity)</td>
</tr>
<tr>
<td>Usefulness</td>
<td>Usefulness to Identify and Correct Errors Overall Development Skills Improvement</td>
<td></td>
</tr>
</tbody>
</table>
3.5 Research Approach

3.5.1 Overview
The objective of this section is to identify ways to develop an error prevention approach using a catalogue of errors, which constitutes the principle objective of this thesis. In order to address the research objective three research questions were identified (section 3.2). The successful investigation of the research questions leads to the accomplishment of the research objective. It follows that the identification of how each individual research question is addressed would lead to the identification of how the research objective will be accomplished.

The review of literature indicates that the terminology used to refer to the way a research objective and/or question is addressed is quite varied and sometime inconsistent. For instance, Galliers (1990) uses the term approach; Galliers & Land (1987) and Jenkins (1985) use the term methodology; Antill (1985) uses the term method; Hamilton & Blake (1982) use the term strategy; Zelkowitz & Wallace (1996; 1997; 1998) use the term model. Given this variety of terms, for the sake of consistency, it was decided that the following definitions were warranted:

i) \textbf{Methodology}: is used to refer to the way the individual research questions are addressed, and

ii) \textbf{Approach}: is used to refer to the collection of methodologies that are adopted to address a group of related research questions, i.e. an overall research objective.

The relationships between the research objective, questions, methodologies and research approach have been portrayed in figure 3.2.

The review of literature on research methodologies suggests that there are many methodologies available. In order to be able to make an informed decision about what methodologies could be the best options to address each research question (see figure 3.2), it is important that the existing options be examined.
The categories of the existing research methodologies are examined first. These methodologies are generally represented by two broad categories, namely, the positivist and the interpretivist methodologies. These categories are covered in section 3.5.2. The aim of section 3.5.3 is to examine the existing methodologies of research and to critically evaluate them in terms of their respective advantages and disadvantages. A number of works have been examined for this purpose. They include the works presented in Antill, 1985; DeLone & McLean, 1992; Galliers, 1990, 1992; Galliers & Land, 1987; Jarvenpaa, 1988; Jarvenpaa, Dickson, & DeSanctis, 1985; Jenkins, 1985; McDermid, 1998; Orlikowski & Baroudi, 1991; Pervan & Klass, 1992; Shanks, Rouse, & Arnott, 1993; Zelkowitz & Wallace, 1996, 1997; Zelkowitz & Wallace, 1998; Zikmund, 1994. For consistency, the work of Shanks, Rouse, & Arnott (1993) is used here as the principal reference. Shanks et al. (1993) draw on the work of Galliers (1992; 1990) and their own experience and argue that research methodologies include laboratory and field experiment/field study, simulation, survey, case study, interpretive study, conceptual/argumentative/subjective study, and action research. In section 3.5.4, the methodologies to address the individual research questions of this thesis are identified and the research approach is defined.
3.5.2 Categories of Research Methodologies

In their paper, Shanks, Rouse, & Arnott, classify research methodologies into two categories. The first category is known by the names of positivism, scientific, empirical; the second category is known by the names of interpretivism, subjective, non-empirical. For convenience, in the following discussion, the two categories are referred to by the terms: positivism and interpretivism. A widely accepted description suggests that the positivist methodologies attempt (Burrell & Morgan, 1979):

"...to explain and predict what happens in the social world by searching for regularities and causal relationships between its constituent elements." (Burrell & Morgan, 1979)

Thus, positivist methodologies attempt to explain the phenomenon under investigation objectively and systematically, rigorously and rationally through empirical investigation. Positivist methodologies also require replication of the investigation by other researchers as the ultimate test for a phenomenon’s explanation (Shanks et al., 1993).

The proponents of the interpretivist methodologies maintain that researchers can never be objective in their explanations of a phenomenon. Therefore, interpretivist methodologies involve the possibility of many different subjective interpretations of the same phenomenon; the impact of the investigator on the phenomenon under investigation; and the context-related problems affecting the predictive ability of future recurrences of the same phenomenon (Galliers, 1990; Shanks et al., 1993). Interpretivist methodologies collect empirical data, but unlike their positivist counterparts, the collected data is subjective or qualitative in nature (Shanks et al., 1993). Finally, interpretivist methodologies are not concerned with the replication of the explanation of a phenomenon, on the grounds that it is impossible to reproduce all factors of the phenomenon under investigation. The proponents of interpretivist methodologies are concerned with the extent to which an explanation is sensible in the context where the phenomenon is investigated (Neuman, 2000).

Shanks, Rouse, & Arnott (1993) advocate that while the categorisation of some methodologies is straightforward, others can belong to either category. For instance, on the one hand, laboratory and field experiments belong to the positivist category because
they are empirical in nature. Similarly, a conceptual study falls into the interpretivist category because it is non-empirical, and involves subjective articulation about a phenomenon or research question or objective. On the other hand, surveys, depending on the nature of the collected data, can belong to both categories. Because of this, Shanks, Rouse, & Arnott (1993) categorise research methodologies on a continuum rather than on two distinct categories. Figure 3.3 was adapted from (Shanks et al., 1993) to present the continuum of the research methodologies discussed above.

<table>
<thead>
<tr>
<th>Field/ Laboratory Experiment</th>
<th>Simulation</th>
<th>Survey</th>
<th>Case Study</th>
<th>Action Research</th>
<th>Interpretive Study</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Positivist</td>
<td>Scientific</td>
<td>Empirical</td>
<td>Interpretivist</td>
<td>Subjective</td>
</tr>
</tbody>
</table>

Figure 3.3 – A continuum of research methodologies

In order for the investigation of the research questions to be comprehensive both positivist and interpretivist methodologies have been used. This has enabled the investigator to capture complementary aspects of the use of the Catalogue of Errors in software development. For instance, when the error density of code artifacts produced by two developers in two development exercises is compared, such that one developer uses a catalogue of errors, whereas the other does not use one, a better explanation of differences suggested by the objective empirical error density data would be gained if, subjective evidence using interpretivist methodologies were obtained as well. For example, a subjective comment like: "developing a code artifact without a catalogue of errors was easy, but once I spent some time to learn about errors in the catalogue, developing a code artifact was even easier and I managed to identify and correct many errors.", would help resolve any possible contradictions concluded by the analysis of the objective error density data alone. In the work of Moher & Schneiderman (1982), where different programming constructs, techniques or tools were compared, a similar argument was made.
3.5.2.1 Relationships Between Research Questions and Methodologies - Revisited

As shown earlier, research questions one and two entail the development of an Error Framework and a Catalogue of Errors. Research question three entails the evaluation of the impact of the Catalogue of Errors on developers' ability to prevent errors as well as their productivity. In this context, research question one and two are conceptual design questions. Question three, on the other hand, is an empirical implementation question. It follows that the methodologies used to address question one and two, should be interpretivist in nature, whereas the methodology used to address question three should be positivist as shown in figure 3.4.

Figure 3.4 – Relationships between research objective, questions, methodologies, and approach

3.5.3 Review and Evaluation of Research Methodologies

In this section several research methodologies are reviewed. This was carried out because the knowledge of the details of individual research methodologies available would help choose the right research methodology. The section concludes with an evaluation of the reviewed research methodologies and their relevance to the research questions that are investigated in this thesis. This includes a short-list of the research methodologies that can be used to address the research questions outlined above. This shortlist is comprised of the field study, and the field experiment methodologies.
Laboratory Experiment
With this methodology human subjects are commonly employed in a controlled environment, mainly a laboratory setting, in an effort to discover precise causal relationships between selected variables. Laboratory experiments are characterised by attempts to exert control on some variables (also known as the independent variables) in order to manipulate other variables (also known as the dependent variables) while systematically and quantitatively measuring them. Quantitative analytical techniques are used to process the measured values of the dependent variables in order to confirm or refute the hypothesised causal relationships between independent and dependent variables (Galliers, 1990; Jenkins, 1985; Zelkowitz & Wallace, 1996, 1997; Zelkowitz & Wallace, 1998). Laboratory experiments are also called *in vitro* experiments (Basili, 1996a). The use of laboratory experiments is regarded advantageous because it provides the researcher with the opportunity to isolate and control a small number of factors and study them intensively. The downside, however, is that laboratory experiments generate results that cannot be readily generalised to the real world. This occurs due to the fact that in the real world, identified relationships are far more elaborate than their laboratory counterparts. In addition, it is often said that since most laboratory experiments employ students as surrogates for true developers, this is sometimes debatable, if plausible generalisations to real world situations are to be made.

Field Experiment/Study
In principle, field experiments are similar to and considered an extension of laboratory experiments (Galliers, 1990). They differ in the fact that field experiments are conducted in a more realistic environment, precisely, in an organisation's natural setting involving actual stakeholders or developers (Galliers, 1990; Jenkins, 1985; Zelkowitz & Wallace, 1996, 1997; Zelkowitz & Wallace, 1998). In addition, field experiments measure the effects of the independent variables on the dependent variables through systematic observation of the human subjects, which may consist of interviews, questionnaires, participant observation etc. Field experiments are also known as *in vivo* experiments (Basili, 1996a). The principal advantage of field experiments is greater generalisability to real world situations. It is however, difficult to find organisations willing to be involved and replications are virtually impossible (Galliers, 1990).
McDermid (1998) suggests that the term field study (also known as field trial) is used to refer to studies which focus on the qualitative (i.e. subjective) rather than quantitative (i.e. objective) aspects of the research questions under investigation. A field study would, therefore, need to be located on the right hand side of Shanks et al.'s (1993) continuum (figure 3.3), due to its interpretivistivistic nature. A field study does not require the definition of hypotheses, which need to be tested in order for causal relationships between/among variables to be confirmed or refuted. With a field study, the investigator of the research question remains the expert providing an answer to the research question under investigation; participants in the field study are considered as components of the experiment, because their feedback about the answer of the research question provided by the investigator is solicited from them; such feedback is subsequently incorporated into a further refinement of the answer of the research question (McDermid, 1998).

Simulation
Simulation is used to study problems where the constituent variables are known, and the behaviour resulting from their interactions is not. A model of the behaviour is constructed and the behaviour is observed as the variables are adjusted. The main problem inherent in this type of research methodology is the fact that it is hard to build a model that reflects the real world, or if such model is built it is hard to know if it is accurate.

Survey
A survey is otherwise known as opinion research and uses questionnaires or interviews to collect data on opinions, attitudes etc. at a particular point in time. The data is analysed using statistical methods so that inferences can be made. The principal advantage of surveys is that depending on the design of the questionnaires or interview questions, they allow for a large number of variables to be taken into consideration. In addition, depending on the allotted budget they may be administered to a reasonably large sample of the population, making the inferred results more generalisable. Despite these advantages, surveys have their weaknesses too. For example, the researcher has no control of who actually completes the questionnaire and there is no way to guarantee that the respondents willing to participate are not biased in their answers (Galliers, 1990; Galliers & Land, 1987; Jenkins, 1985).
Case Study

A case study is an empirical inquiry that attempts to investigate a problem in its real-life context. Typically, in a case study the boundaries of the problem are not always clearly evident. Consequently, data are collected using a variety of means, such as interviews, questionnaires, observation etc. (Gable, 1992). Galliers (1990) suggests that the main advantage of the case study is that it allows "the capture of 'reality' in considerably greater detail (and the analysis of a considerably greater number of variables)" (Galliers, 1990, p. 162) as opposed to the previously discussed methodologies. The pitfall is that a case study is normally confined to a single problem in an organisation, which may be difficult to reproduce in other similar organisations, in order to obtain statistically significant results (Galliers, 1990).

Action Research

Action research is often construed as a special case of the case study methodology (Galliers, 1990). In action research, a problem or research question is investigated with the active and reflective participation of the researcher (often in a consultant role). At the completion of action research the role and the impact of the research is identified and acknowledged (Baskerville & Wood-Harper, 1996; Wood-Harper, 1985). The main strength of action research lies with the fact that the knowledge that is likely to accrue from action research may be immediately applied in the organisation where action research takes place. In addition, the researcher's involvement is overtly acknowledged, helping clarify the researcher's biases. The main weakness of action research is similar to what was identified for the case study methodology.

Interpretative Study

Interpretative study involves the articulation of the essence of a phenomenon, in terms of meaning, preconditions, prejudices and assumptions (Shanks et al., 1993). The essence of a phenomenon is not verified empirically, because the proof of the essence of a phenomenon is "its self-evidence" (Galliers, 1990, p. 165). An interpretative study is mostly used to understand existing problems rather than investigate new ones (McDermid, 1998). The main strengths of interpretative study is the depth and the richness of the understanding that the researcher can gain about the phenomenon under investigation. The main weakness relates to the "inability to exclude alternative explanations" (Shanks et al. 1993, p. 39) and the researcher's bias (Galliers, 1990).
**Conceptual/Argumentative/Subjective Study**

Shanks, Rouse, & Arnott (1993) define a conceptual/argumentative/subjective study as “the articulation of subjective beliefs about an area of investigation.” (p. 39). This methodology can be used to review existing bodies of knowledge by conducting an in-depth analysis of the existing literature (Shanks et al., 1993). The results of such analysis can provide a basis for further empirical studies. The principle strength of this methodology is that it allows for a critical analysis of existing knowledge, which can lead to deeper understanding, development of theories and new insights (Shanks et al., 1993). The main weakness is that it can be highly subjective and subject to bias and distortion (Shanks et al., 1993).

**Evaluation of Research Methodologies**

Shanks, Rouse, & Arnott (1993) suggest that the nature of the research question determines the choice of research methodology. This implies that some methodologies can be more suitable than others to tackle certain types of research. In this context, three types of research can be distinguished. Exploratory research is typically used to explore existing literature and help discern generic models, frameworks, and insights (Shanks et al., 1993). Descriptive research attempts to accurately describe the specific structural and functional details of a setting (Shanks et al., 1993). Explanatory research attempts to investigate why things happen the way they do (Shanks et al., 1993).

Table 3.2 was adapted from (Shanks et al., 1993) and summarises which research methodologies are best suited for each type of research defined above. Shanks et al.’s (1993) table has been augmented with the field study entry. A field study helps explore a phenomenon by soliciting subjective or interpretivist feedback by the stakeholders involved in the phenomenon. A field study may help explain causal relationships but not absolutely, because feedback from participants will be subjective. A field study may also help describe the specific structural and functional details of a phenomenon, provided that the stakeholders have been involved in the phenomenon under investigation.
Table 3.2 - Use of Different Methodologies to Research

<table>
<thead>
<tr>
<th>Research Type</th>
<th>Lab/Field Experiment</th>
<th>Simulation</th>
<th>Survey</th>
<th>Case Study</th>
<th>Action Research</th>
<th>Field Study</th>
<th>Interpretive Study</th>
<th>Conceptual Study</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploratory</td>
<td>No</td>
<td>Yes</td>
<td>Maybe</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Descriptive</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Maybe</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Explanatory</td>
<td>Yes</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
<td>No</td>
<td>Maybe</td>
<td>Maybe</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 3.2 may be helpful to narrow down the choice the research methodologies to be adopted given the type of research that the research questions discussed at the beginning of this chapter represent.

The first question is an attempt to discern a generic Error Framework by exploring the literature to identify what type of information may be helpful to developers in learning about errors. This suggests that this question entails exploratory research. Similarly, the second question seeks the exploration of literature for information in order to develop a Catalogue of Errors, based on the Error Framework. Hence, the second question entails exploratory research as well. Finally, the third question is an attempt to investigate whether a causal relationship exists between the knowledge of a Catalogue of Errors, and the number of errors injected by developers into software development artifacts and the productivity with which such artifacts are developed. This suggests that the third question entails explanatory research. In this context, the best research methodologies suitable to address the three research questions have been summarised (table 3.3).

Table 3.3 - Potential research methodologies to use to address research questions

<table>
<thead>
<tr>
<th>Research Type</th>
<th>Research Question</th>
<th>Research Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploratory</td>
<td>Question 1, 2</td>
<td>Simulation; Case study; Action research; Field Study</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Interpretive study, Conceptual study</td>
</tr>
<tr>
<td>Exploratory</td>
<td>Question 3</td>
<td>Laboratory/Field experiment</td>
</tr>
</tbody>
</table>

Simulation and laboratory experiment can be deleted from table 3.3, because they lack the real-world context. Similarly, conceptual study and interpretative study do not involve any kind of testing involving participants (McDermid, 1998). As indicated by the elaboration of the success factors for the research questions (see section 3.4), research question one and two require participant feedback concerning the Error Framework and the Catalogue of Errors. According to McDermid (1998) a case study
“involves the observation of an existing phenomenon in a relatively unobtrusive way” (p. 97). The results of the investigation of question one and two, however, are meant to be enriched with feedback from participants. Therefore, case study is not a good choice to investigate them. Action research requires the active participation of the researcher. Question one and two do not require such participation. Thus, action research can be removed as well. Having found that some of research methodologies from table 3.3 are not suitable to address the questions under investigation, a short-list has been produced in table 3.4.

<table>
<thead>
<tr>
<th>Research type</th>
<th>Research Question</th>
<th>Research Methodology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploratory</td>
<td>Question 1, 2</td>
<td>Field Study</td>
</tr>
<tr>
<td>Exploratory</td>
<td>Question 3</td>
<td>Field experiment</td>
</tr>
</tbody>
</table>

Obviously, table 3.4, has narrowed down the search for methodologies to addresses the research questions significantly. A more detailed analysis to identify and define the research approach to be used in this thesis is carried out in the following section.

3.5.4 Identifying the Research Approach

The objective of this section is to explain the research approach adopted in this thesis (see figure 3.5). In order to address research question one, which requires the development of an error framework, the existing literature must be reviewed and analysed. The outcome of the literature review and analysis should culminate with the proposal of an Error Framework, which must be empirically validated as suggested by question one success factors (section 3.4.1). This methodology of research fits McDermid's field study definition (McDermid, 1998). A similar methodology can be adapted to address research question two, which requires the development of a Catalogue of Errors, i.e. review and analyse the literature and empirically validate the outcome of the review and analysis using question two success factors (section 3.4.2).

Finally, in order to address question three, which seeks to evaluate the impact of the Catalogue of Errors on the number of errors committed by developers and their productivity a field experiment can be used (table 3.4). Note that figure 3.5 is an evolved version of figure 3.4, which was discussed earlier.
Clearly, figure 3.5 suggests that a hybrid research approach has been adopted to address the three questions. This approach consists of two separate field studies whose objectives are to address questions one and two, and a field experiment which addresses question three. Note that in figure 3.5 the overall research approach has been named as a field study. It is accepted that this name does not exactly fit the description of the field study methodology described earlier in the chapter (see section 3.5.3). However, given
the need to name the research approach of this thesis, it is believed that the term field study is the only term that does justice to nature of the overall research approach portrayed in figure 3.5.

The research approach constitutes a study about errors and the impact that the knowledge about them can have on the number of errors injected in software development artifacts and developer productivity. This study took place in a field setting. In order to avoid any possible confusion in the following chapters the research approach will be referred to as the field study approach, whereas the methodologies used to address question one and two will otherwise be generically referred to as field study methodologies.

3.5 Summary
This chapter has focused on four main issues. First, the research objective and three research questions were formulated in section 3.2. The three questions are to be addressed sequentially and comprise the development of an Error Framework (research question one), which is used as a template to develop a Catalogue of Errors (research question two). The Catalogue encompasses errors from the entire development lifecycle. The Catalogue of Errors should be incorporated into software development and used to help developers understand and learn about potential errors that they are likely to introduce into their development artifacts. The impact of the Catalogue of Errors on the number of errors injected by developers into development artifacts and their productivity must be identified and quantified (research question three). The chapter continued with section 3.3, which argued about the need and the potential benefits of addressing each research question. In section 3.4, the factors that would determine the success of each research question were addressed. Finally, in section 3.5, the research approach was defined. This section has also presented a review of the categories of various research methodologies and has evaluated them according to accepted research criteria. Section 3.5 has culminated with the conclusion that the research approach should be a hybrid approach consisting of two field studies and a field experiment. The issues related to the design of the field studies and the field experiment, context selection, etc. are addressed in more detail in chapter four.
4. Research Design Issues

4.1 Overview

In chapter three the research approach to address three research questions was identified. This research approach is comprised of two field studies and a field experiment. The objective of the first field study is to define and empirically evaluate the Error Framework (research question one). The objective of the second field study is to use the Error Framework in order to develop the Catalogue of Errors and empirically evaluate it (research question two). The objective of the field experiment is to empirically evaluate the impact of the Catalogue of Errors on the ability of developers to prevent errors or to detect and correct them early in the development and their productivity (research question three). While chapter three has provided a bird's eye view of the research approach, it has ignored a number of specific research design issues that are related to the mechanics and the technical aspects of its practical implementation.

For example, issues relating to the context and participant selection with respect to whom the research questions are investigated must be addressed. Unlike field experiments, field studies do not require the definition of hypotheses to be tested in order to confirm or refute causal relationships between variables (McDermid, 1998). However, field experiment design issues (e.g. hypotheses definition) must be meticulously considered and adequately resolved. In both field studies and field experiments the appropriate data collection instrument design issues must be considered in order to ensure that participant feedback is captured accurately. Participant feedback is represented by the data collected from participants using data collection instruments. In both field studies and field experiments, participants must be carefully prepared before their feedback is elicited and the collected data must be adequately validated before any plans for their analysis are carried out. The collected data must be summarised and analysed in order to discern patterns that help answer the research questions. Therefore, data analysis methods and presentation issues must be planned for.

Issues like the ones that were described in the previous paragraph require careful consideration and adequate resolution, before the research approach can be carried out.
in practice. These issues have been collectively named as research design issues and they are addressed in this chapter.

The research design issues discussed in this chapter have been categorised into three major groups:

i) research planning issues,

ii) research operation issues, and,

iii) data processing and presentation issues.

Table 4.1 summarises the individual research design issues under their respective categories and also indicates their relevance to the research questions. This is important not only because it provides an overview of the chapter but also because it shows that the relevant research design issues have been addressed for each research question.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Context Selection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Participant Selection</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Design of Field Experiment</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Instrumentation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Research Operation Issues</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Preparation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data Validation</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Data Processing and</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Presentation Issues</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

This chapter culminates with the resolution of several important design issues and the presentation of the supporting arguments. Firstly, a university environment is selected as a research context and the rationale for the decision to use senior student developers as participants in the research is explained. Secondly, the rationale for using a non-randomised field experimental design to investigate question three is presented. Thirdly, the chapter continues with the development of a hierarchy of data collection instruments in order to capture participant feedback with regard to research questions. Fourthly, the need for participant commitment and data validation is explained. Finally, the chapter concludes with the discussion of the plans for data analysis and presentation.
4.2 Research Planning Issues

The research planning issues have been divided into four major groups. Context selection issues, participant selection issues, experimental design issues, and instrumentation issues.

4.2.1 Context Selection

The research was carried out in a university environment in the context of a unit (US equivalent of 'course') called the Internet and Java Programming taught in Edith Cowan University, Perth, Western Australia. This unit is a final year advanced programming unit and is offered to senior computer science and software engineering undergraduate students. The enrolled students are expected to develop a major non-trivial simulator project in the Java programming language. The project is given to students in an unstructured formal problem statement at the beginning of the semester. In order to successfully complete the simulator project, the students need to develop three separate, but related software development artifacts, which constitute progressive refinements of the problem statement into a requirements artifact, design artifact and code artifact for the underlying simulator project over a period of 12 weeks in a 13 week semester.

The Internet and Java Programming unit is offered at two separate campuses of the university, namely, the Mount Lawley Campus and the Joondalup Campus. There are no differences in the way the unit is offered and run in both campuses (e.g. the same lecturer taught on both campuses). In addition, the unit was offered in both semesters of 2001. In order to ensure that students did not use their seniors' projects as their own, the only part of the unit that was changed was the problem statement for the simulator project. In Semester 1 2001, the problem statement of an Intersection Simulator was provided, whereas in the following semester the problem statement of a Hotel Lobby Simulator was given. The problem statements for semester 1 and 2 simulator projects have been provided in appendix B, sections 2.1.1 and 2.1.2.

The same delivery dates applied for development artifact submission for students enrolled in both campuses and in both semesters and were Week 4, Week 8 and Week
12, for the requirements, design, and code artifacts, respectively. This arrangement is summarised in table 4.2

<table>
<thead>
<tr>
<th>Project</th>
<th>Semester 1 2001 Mount Lawley</th>
<th>Semester 1 2001 Joondalup</th>
<th>Semester 2 2001 Mount Lawley</th>
</tr>
</thead>
<tbody>
<tr>
<td>No of Students</td>
<td>113</td>
<td>39</td>
<td>67</td>
</tr>
<tr>
<td>Week 4</td>
<td>Requirements Artifact due</td>
<td>Requirements Artifact due</td>
<td>Requirements Artifact due</td>
</tr>
<tr>
<td>Week 8</td>
<td>Design Artifact due</td>
<td>Design Artifact due</td>
<td>Design Artifact due</td>
</tr>
<tr>
<td>Week 12</td>
<td>Code Artifact due</td>
<td>Code Artifact due</td>
<td>Code Artifact</td>
</tr>
</tbody>
</table>

The principal benefits associated with the selection of this context are related to the saved effort that is associated with participant selection, research approach set up, replication etc. Such issues are discussed in more detail later in this chapter. Since the selected context directly affects the ability to generalise the findings to a wider population, issues concerning threats to generalisation etc. are addressed in detail in chapter five where validity is evaluated.

### 4.2.2 Research Approach Revisited

The fact that two semesters and three groups of participants were available presented the opportunity for replication. The first trial of the study took place during semester 1, 2001, and the replication took place in semester 2, 2001. The replication was seen as an important way to confirm or refute the results obtained from the first trial. This arrangement is summarized in figure 4.1. A further discussion on replication benefits is provided later in this chapter.

### 4.2.3 Participant Selection Issues

This section contains an overview of various participant selection issues including, population definition, sampling, sample size definition, and the types of participants determination relating to this research. The conclusion is that student participants can be a valuable participant selection pool.
Figure 4.1 - Research Approach Revisited
4.2.3.1 Population and Sampling

A population consists of the totality of the observations with which the investigator is concerned (Walpole & Myers, 1993). Usually it is not feasible to deal with every individual of the population, which suggests that a sample needs to be drawn from the population of interest. A sample is, therefore, a selection from or a subset of the target population. The sample definition is very important because it not only determines the ways in which sample data may be analysed, but also it determines the various facets of the validity (e.g. generalisation) of the research. Two main categories of samples are recognised. Probability or representative samples presume that the probability of the selection of each participant is known before the sample is drawn, whereas, non-probability samples presume that probability is unknown (Robson, 1993).

There are many types of probability sampling techniques. The most popular ones include simple random sampling, systematic sampling, and stratified random sampling. Simple random sampling involves selecting participants at random from the target population (Moher & Schneider, 1982). Systematic sampling requires choosing a random starting point in the intended sample and then selecting the nth occurrence as many times as the required sample size (Robson, 1993). Stratified random sampling entails dividing the populations into a number of groups or strata and then applying simple random sampling within each stratum (Moher & Schneider, 1982). Robson (1993) points out that probability samples can be used to make statistical inferences about the populations. He, however, also says that:

"The exigencies of carrying out real world studies can mean that the requirements for representative sampling are very difficult, if not impossible, to fulfil. [representative or probability] sampling frames may be impossible to obtain" (p. 142).

This implies that that while probability sampling is feasible in principle, it also presents extremely difficult practical and ethical issues (Robson, 1993).

There are many types of non-probability sampling techniques. The most popular ones, include, quota sampling and convenience sampling. Quota sampling requires the

---

13 The impact of sampling on data analysis is examined later in the chapter. The impact of sampling on validity is examined in chapter five.
acquisition of readily available representatives of the various elements in the target population, normally in the relative proportions in which they occur (Robson, 1993). With convenience sampling, participants are chosen on the basis of who is readily and conveniently available (Moher & Schneider, 1982). Robson (1993) says that non-probability samples are easier to obtain. In general, the non-probability samples can be used to make sensible judgements about the population, but such judgements cannot be made on strong statistical grounds.

From the above discussion it is concluded that in a field study or experiment the population and sampling technique must be determined.

4.2.3.2 Sample Size
The number of participants is another issue that requires consideration. The more participants take part in empirical research, the better the chances that the experiment will yield meaningful results. Moher & Schneider (1982) advocate that as more participants get involved in an experiment exercise, tighter confidence intervals to determine the significance of results become available to the investigator. In addition, the larger the number of the participants, the lower the probability of making incorrect conclusions that the sample findings can be applied to the population as well (Miller, Daly, Wood, Roper, & Brooks, 1997). This implies that in a field study or experiment, the size of the sample size must be adequately determined.

4.2.3.3 Professional versus Student Participants
The review of literature suggests that in general two types of participants can be identified. First, professional participants are participants who work in the software development industry. Second, novice or student participants, include participants who are enrolled in software development-related university degrees (e.g. software engineering, computer science, etc.). There are two distinct schools of thought that advocate which type of participants should be used in empirical studies (e.g. field studies and experiments). The proponents of the first school of thought suggest that novice developers are no different from the experienced professional developers. The proponents of the second school of thought object to such an idea by advocating that the two types of developers, i.e. novice and professional, are quite different and must,
therefore, not be used to replace each other. Referenced citations of the proponents of these two schools of thought are provided below.

Moher & Schneider (1982) quote Miller defending the employment of participants with minimal experience or novice developers:

"Use of naïve subjects [participants] over experienced programmers may permit detection of the characteristics of those processes unconfounded by differential training, experience, expectations, and attitudes associated with professionals." (p.68).

Supporters of the first school of thought not only promote the idea conveyed by Miller's statement, but they also go one step further and suggest that results of such experiments may even be generalised to a broader range of developers, including experienced professional developers.

In the same discussion, Moher & Schneider (1982) quote Young, a proponent of the second school of thought, who argues that the two categories of developers are quite different:

"Overall, it appeared that advanced programmers committed errors of different types more equally, eliminated the superficial ones more rapidly, and thus accounted for fewer errors in total. Beginners were apparently more subject to the commission of some errors than others, but proceeded to eliminate all types of errors with nearly the same facility." (p. 69).

Weinberg (1998) brings an interesting dimension into the discussion. This dimension suggests that it is the amount of exposure on the tool or methodology etc. that is being empirically evaluated or experimented with, which distinguishes between professional and student developers. This amount of exposure is what determines experience:

"In our study of JCL (Job Control Language), some of the students were relative novices at programming, and others were rather old hands. All, however, were novices at JCL. ... When the group was tested at the beginning of the course, experienced and inexperienced alike did poorly; at the end of the course, the groups did equally well. Thus it was not [programming] "experience" in general
that mattered in this problem, but the specific experience with JCL. "(Weinberg, 1998, p.34).

In another study, Brooks (1999) quotes Young who concludes about his study on error rates that:

"... the study of error rates in programming which revealed, surprisingly, that experienced programmers initially make about the same number of errors as beginning programmers, but are able to find their errors faster." (p.198) (Brooks, 1999).

This suggests that errors in programming are committed in the same way by both professional and novice programmers. These two categories of participants, differ in the way they detect errors, however.

The study conducted by Host, Regnell, & Wohlin (2000) presents an empirical evaluation of the differences between student and professional software developers in non-trivial tasks that require knowledge and understanding of professional software development. Host, Regnell, & Wohlin (2000) were unable to detect any significant difference between the two selected groups, leading to the conclusion that in empirical studies, software engineering students may be used instead of professional software developers. They, however, are cautious not to generalise this conclusion to all possible situations, conditions and tasks (Host, Regnell, & Wohlin, 2000). The obvious conflict between the above two schools of thought, and the arguments presented by Weinberg (1998), Host, Regnell & Wohlin (2000), and Brooks (1999), lead one to a straightforward conclusion. It is possible to use student participants in empirical studies focusing on how errors are committed. However, extreme care and strong justification is needed when the empirical results obtained from the student participants are generalised to professional developers.

4.2.3.4 An Evaluation of Participant Selection Issues to Identify Participants

In this section, the participant selection issues that were reviewed above are discussed with reference to the context where the research questions (see chapter three) are investigated.
Population and Sampling

The population used in this research was that of third year advanced Java student developers enrolled in computer science and software engineering degrees in Edith Cowan University (ECU), Western Australia. As Wohlin (1998) suggests, third year students are about to enter the software development industry, therefore, the population included junior Java developers who are about to or have just joined the industry.

Given the context, the probability of selection of the participants is not known in advance, therefore, non-probability sampling was used. In addition, the participants were readily and conveniently available because they had signed up for a unit which means that the sample was a convenience sample (Wohlin, 1998; Wohlin et al., 2000). Statistics literature suggests that convenience samples are not recognised to produce representative findings (Robson, 1993). Robson (1993), however, reports Bryman (1989) showing only a few instances of probability samples being used in the research literature, while according to Schwab (1985):

"... almost all of the empirical studies published in our journals use convenience, not probability samples... Thus if one took generalisation to a population using statistical inference seriously, one would recommend rejecting nearly all manuscripts submitted." (p.173).

Examples of studies using convenience sampling may be found in Wohlin, (1998); Wohlin et al., (2000). These studies suggest that extreme caution needs to be taken when generalisations to the target population are made with convenience sampling.

Number of Participants

The participants were selected from the students who were enrolled in the Internet and Java Programming unit. The number of participants was determined by three factors: 1) number of students that enrolled in the unit; 2) number of students that allowed investigators to use their results; 3) number of students that decided to withdraw from the experiment without completing all development artifacts. Table 4.3 distinguishes between participants who began the empirical study and those who completed it. Nine participants did not complete all required development artifacts. The impact of the size of the three participating groups on statistical significance is revisited later in chapter five, where the validity of the research is examined.
Table 4.3 - Sample size of the three participating groups

<table>
<thead>
<tr>
<th>Campus</th>
<th>Semester/Year</th>
<th>No of Participants At start</th>
<th>No of Participants At Completion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mount Lawley</td>
<td>1/2001</td>
<td>133</td>
<td>126</td>
</tr>
<tr>
<td>Joondalup</td>
<td>1/2001</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>Mount Lawley</td>
<td>2/2001</td>
<td>67</td>
<td>65</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>239</td>
<td>230</td>
</tr>
</tbody>
</table>

**Type of Participants**

Numerous experimental studies employing students as participants were surveyed (Battig, 1998; Host et al., 2000; Land et al., 1997; Porter et al., 1995; Schneider et al., 1992; Shull, 1998; Stone et al., 1990; Wohlin, 1998; Wohlin et al., 2000). All these studies converged on the reasons for selecting students rather than professionals as participants. First, while there is no doubt that professional developers are the ideal participants, it is also recognised that it is very difficult to engage them into experimental exercises without remuneration because they must forgo their existing commitments. An alternative was to seek professional developers who would be willing to participate in the experiment while retaining their commitments. This, however, would have the potential to marginalise the study, because the participants' own commitments would carry a higher priority. Loss of control over the experiment might ensue. In short, professional developers were found to be unattractive participants, when cost, convenience, and control were considered.

Student participants, on the other hand, were willing to participate in empirical studies when such studies were part of a unit that they were enrolled in. In these circumstances, it is possible to integrate the learning objectives of the unit with the research goals of the study. Most of the time, student participants are readily available and do not have to be paid any remuneration. Besides, if the learning objectives were integrated with the empirical study goals, the study under investigation would be a top priority with the students. In such a case, greater control is achievable. For instance, as shown in table 4.2, the student participants were to complete the development of simulator projects, by delivering requirements, design and code artifacts. These circumstances constituted a very suitable context for the research.
In addition, student participants were more likely to be available in larger numbers than professional developer participants. This was expected to boost the significance of the results. Another argument favouring the selection of student participants was the benefit of conducting an empirical study in a university environment. If positive outcomes were obtained, it would be possible to replicate the empirical study in the industry. If negative outcomes were obtained, there would be no costs to the industry. Therefore, initial studies conducted in a university environment may determine whether further research with professional developers is warranted (Linkman & Rombach, 1997).

Another final argument supporting the use of students as participants is that empirical evaluation (i.e. experimentation) can become an integral part of software engineering and computer science education (Wohlin, 1998; Tichy, 1998).

The discussion about the impact of student developers on the generalisability of the findings is covered in chapter five.

4.2.4 Design of Field Experiments
In this section, field experiment issues related to variable definition, experimental hypotheses definition, randomisation, and balancing are examined. Figure 4.1 shows that two field experiments were conducted. The variables and experimental hypotheses definition were identical for both field experiments conducted in semester 1 and 2, 2001. However, some randomisation and balancing considerations were different and where applicable, such differences are highlighted in the following discussion.

4.2.4.1 Variable Definition
In any experiment exercise, two types of variables are recognised. They are called independent and the dependent variables. Wohlin et al. (2000) define an independent variable as any variable that may be manipulated by the investigator of the experiment (Wohlin et al., 2000). Wohlin et al. (2000) also define a dependent or response variable as one that the investigator is interested in and whose values are expected to be affected in response to changes in the independent variable.

The field experiments conducted in this study (see figure 4.1) manipulate one independent variable, namely, Java development, including requirements, design, and
Java coding. This independent variable has two possible values, namely, Catalogue-assisted Java development and standard Java development. In order to avoid any ambiguities that may arise with the two possible values of the Java development independent variable, the following definitions are warranted:

i) *Catalogue-assisted Java Development* — the participants were to use the Catalogue of Errors and training to assist them to develop requirements, design, and code artifacts sequentially;

ii) *Standard Java Development* — the participants were to produce the requirements, design, and code artifacts sequentially.

Such values of the independent variable are otherwise referred to as treatments of the independent variable. A similar approach to the definition of independent variables is applied in Land et al., (1997); Moher & Schneider, (1982); Shull, (1998).

Seven categories of dependent variables have been identified (see section 3.4.3, chapter three). The dependent variables include:

i) Error type/count;

ii) Time spent to develop requirements/design/code artifacts;

iii) Size of requirements/design/code artifacts;

iv) Density of errors;

v) Scope ratios;

vi) Productivity of developers when constructing requirements/design/code artifacts.

Given the definition of the dependent variables, it is worth reiterating that the values of the dependent variables are measured and evaluated in order to assess whether the manipulated independent variable has a statistically significant effect on them. The relationship between the dependent and independent variables and the underlying experiment(s) has been depicted in figure 4.2.
Figure 4.2 – The relationship between Dependent and Independent Variables

4.2.4.2 Hypothesis Formulation

One of the most important aspects of an experiment is to formally and clearly state what the experimental hypotheses are (Wohlin et al., 2000). Before such definition can be made, however, a preliminary experimental design issue must be addressed. This issue is related to the fact that the Java development independent variable, described above, has two possible treatments involving two groups of participants. For convenience and simplicity, the two groups were named and defined as follows:

i) Group A

Group A, the non-intervention group, was to develop software artifacts, namely, requirements, design, and code using the standard Java development treatment of the independent variable (see section 4.2.4.1).

ii) Group B

Group B, the intervention group, was to develop software artifacts, namely, requirements, design, and code using the Catalogue-assisted development treatment of the independent variable, (see section 4.2.4.1).

Having defined the two participating groups, an informal definition of the hypotheses was made as follows:

1. It is hypothesised that group B participants would commit fewer errors in the development of their artifacts as opposed to the participants of group A. The error densities of groups A and B would be measured and compared to confirm or refute the hypothesis. The expected result would be attributed to the fact that group B received error-related information and training with a Catalogue of Errors.
2. It is hypothesised that group B participants would have fewer errors that escape to other phases as opposed to the participants of group A. The escape ratios of groups A and B would be measured and compared to confirm or refute the hypothesis. Again this would be attributed to the fact that group B received error-related information and training with a Catalogue of Errors.

3. It is hypothesised that group B participants would have a lower productivity in Java development as opposed to the participants of group A. Productivity would be measured in terms of the artifact size produced in relation to the amount of time spent to construct the artifact. The amount of time spent to deliver the artifact would not include the time spent learning the Catalogue of Errors. This decision was made because Catalogue learning time occurred only once for each participant.

On the basis of the above informal statements, the hypotheses can be stated formally as below:

1. Null Hypothesis, $H_0$: There is no difference in the density of errors committed, between the participants of group A and B.

   $H_0$: Error Density(group A) = Error Density (group B)

   Alternative Hypothesis, $H_a$: Error Density (group A) $\neq$ Error Density (group B)

Measurements needed:

- Density of Catalogued Requirements Errors; Density of Other Requirements Errors;
- Density of Catalogued Design Errors; Density of Other Design Errors;
- Density of Catalogued Code Errors; Density of Other Code Errors

Given that six different measurements were needed, six different hypotheses were tested. These hypotheses have been summarised in Table 4.4.
2. Null Hypothesis, $H_0$: There is no difference between group A and B in the number of errors that escape to phases other than that of origin.

$H_0$: Escape Ratios (group A) = Escape Ratios (group B)

Alternative hypothesis, $H_1$: Escape Ratios (group A) $\neq$ Escape Ratios (group B)

Measurements needed:
- Escape Ratios of Requirements Errors escaping to Design Artifacts
- Escape Ratios of Requirements Errors escaping to Code Artifacts
- Escape Ratios of Design Errors escaping to Code Artifacts

Given that three separate measurements were required, three different hypotheses were tested. The hypotheses have been summarised in table 4.5.

3. Null Hypothesis, $H_0$: There is no difference in the productivity (measured as the ratio of artifact size with the artifact development time) between group A and B.
H₀: Productivity (group A) = Productivity (group B)

Alternative hypothesis, Hₐ: Productivity (group A) ≠ Productivity (group B)

Measurements needed:
- Productivity to develop Requirements Artifacts
- Productivity to develop Design Artifacts
- Productivity to develop Code Artifacts

Given that three separate measurements were required, three different hypotheses were tested. The hypotheses have been summarised in table 4.6.

<table>
<thead>
<tr>
<th>No</th>
<th>Null and Alternative Hypotheses</th>
</tr>
</thead>
</table>
| 10 | Productivity to develop requirements artifacts (PDRAs): H₀: PDRAs (B) = PDRAs (A)  
Hₐ: PDRAs (B) ≠ PDRAs (A) |
| 11 | Productivity to develop design artifacts (PDDAs): H₀: PDDAs (B) = PDDAs (A)  
Hₐ: PDDAs (B) ≠ PDDAs (A) |
| 12 | Productivity to develop code artifacts (PDCAs): H₀: PDCAs (B) = PDCAs (A)  
Hₐ: PDCAs (B) ≠ PDCAs (A) |

The type of statistical tests used to test these hypotheses is examined in section 4.4.3.

4.2.4.3 Randomization

The aim of this section is to explore the available possibilities for assigning the two treatments of the independent variable to the available participants. Two options were possible:

i) Option one: assign participants randomly, or

ii) Option two: assign participants arbitrarily using the existing segregation (i.e. based on campus location and semester).

**Option One**

The first option is typically used in experiments. Thus, Cook & Campbell (1979) refer to randomised experiments ¹⁴ which "are characterised by the use of initial random assignment for inferring treatment-caused change." (pp. 5-6). However, they point out a

---

¹⁴ Robson (1993) refers to randomised experiments as true experiments.
practical difficulty with randomised experiments in general and with randomised field experiments in particular:

"It is more difficult to assign individuals or larger social groups to treatments at random than it is to assign agricultural plots. It is also more difficult to assign individuals to treatments at random in field experiments than in laboratory settings. The field researcher is often a guest in at the sites where he or she works while the laboratory researcher has almost total control over the setting and acts as the respondent's host. Such considerations imply that random assignment will be less frequent with humans than with objects and less frequent with humans in the field than in the laboratory." (p. 6)

Judd, Smith, & Kidder (1991) add:

"...there are frequently practical and ethical considerations that force the researcher to use some design other than a randomized experimental one." (p. 34).

Furthermore, if participants were to be assigned to the treatments of the independent variable randomly, there would be a high probability that participants assigned to either of the treatments would be sharing the same campus. This would have presented the potential problem of data sharing between the participants. This problem is recognised and discussed by Basili et al. (1998; 1999). Data sharing would have the potential to undermine and invalidate the experiment results for both groups of participants (Basili et al., 1998; 1999). In addition, the experiment requires that the Catalogue-assisted Java development treatment participants be trained with the Catalogue of Errors prior to the software development. The training materials were integrated into the lecture notes for the Internet and Java Programming unit, where the participants were enrolled as students. Therefore, if option one were pursued, four sets of lectures (for two randomly selected groups in two campuses) would have had to be delivered. Therefore, option one would have presented difficulties with randomising treatments and data sharing as well as training materials.

**Option Two**

Option two requires the arbitrary assignment of the treatments of the Java development independent variable to the participants, which means that the experiment would lack
randomisation, raising the validity of the term ‘experiment’. Cook & Campbell (1979) distinguish a category of experiments called quasi-experiments. Robson (1993) quotes Campbell & Stanley (1963) who define a quasi-experiment as:

"a[n experimental] research design involving an experimental approach but where random assignment to treatment and comparison groups has not been used." (p.98).

Cook & Campbell (1979) point out that the investigator of quasi-experiments has the additional responsibility to separate the effects that may be caused by initial non-comparability of the participants from the effects that are caused by the treatment. They describe quasi-experiments as where:

"...the comparisons depend on nonequivalent groups that differ from each other in many ways other than the presence of a treatment whose effects are being tested. The task confronting persons who try to interpret the results from quasi-experiments is basically one of separating the effects of a treatment from those due to the initial noncomparability between the average units in each treatment group; only the effects of the treatment are of research interest. To achieve this separation of effects, the researcher has to explicate the specific threats to valid causal inference that random assignment rules out and then in some way deal with these threats. In a sense, quasi-experiments require making explicit the irrelevant causal forces hidden within the ceteris paribus of random assignment.” (Cook & Campbell, 1979, p. 6)

Thus, the quasi-experimental approach, while it liberalises the randomisation restriction and failing to rule out threats to the incomparability among participants, it shares all the principal aspects with the true experiment including hypothesis testing, cause and effect investigation, validity assessment etc. (Robson, 1993). However, the comparability among participants is very important, because its absence can mask any variability of the dependent variables resulting from the manipulation of the independent variable (Moher & Schneider, 1982).

There are “literally thousands” (Weinberg, 1998, p. 35) of factors that can affect the comparability among participants (Keppel, Saufley, & Tokunaga, 1992). Strictly speaking, the number of such factors is “infinite and unspecified” (Robson, 1993, p. 47)
and even randomisation cannot totally control them. Randomisation only renders the effects of such non-treatment related factors on the comparability of the participants, implausible to a specified probability (Keppel et al., 1992; Robson, 1993). It follows that in quasi-experimentation a reasonable list of factors that is expected to have an effect on the comparability of the participants, and hence on the dependent variables, must be specified and controlled for. In this study these factors will henceforth be collectively referred to as the development background independent variable. For practical reasons, the remaining factors, i.e. non-development background related factors, have to be assumed at a fixed level for all participants. Hereinafter, such factors are referred to as other independent variables.

The literature suggests that there are two common ways to capture information about the development background independent variable that affects the comparability of the participants. One way involves the pre-testing of participants and the collection of pre-test data. The second way involves the collection of supplementary evidence (e.g. participant development background data via questionnaires) about possible non-treatment causes of variation in the dependent variables (Frankfort-Nachmias & Nachmias, 1992; Judd, Smith, & Kidder, 1991). Either way, the collected data may be used to help identify threats that may result in non-treatment-caused effects on the dependent variable(s) and, therefore, establish participant comparability. Pre-testing is suggested by Robson (1993), whereas questionnaires have been used by Wohlin (1998), Schneider, Martin, & Tsai (1992), (Frankfort-Nachmias & Nachmias, 1992; Judd et al., 1991) etc. In short, quasi-experiments are the best choice to consider when it is not possible or when it is difficult to randomise the allocation of the treatments of the independent variable (Judd et al., 1991; Robson, 1993).

**Identifying the Option to be Applied**

One advantage of using option two in the context of this research is related to the fact that the participants taking the different treatments would be located in two different widely separated campuses. This would minimise the probability of data sharing between participants. The other advantage of option two is that, if pursued, only two sets of lectures would be required for delivery, resulting in relatively lower cost.
The selection of option two, however, requires an additional decision to be made with regard to the use of pre-testing or questionnaires in order to capture supplementary information.

The use of pre-testing has two problems. Firstly, given the scope of the project (i.e. Intersection and Hotel Lobby Simulators) that the participants were expected to develop as part of the field experiments (a 12 week project in a 13 weeks semester duration), there was insufficient time to pre-test participants with meaningful pre-test task(s) that would reveal useful comparability data. Secondly, if it is assumed that such meaningful pre-test task(s) could be identified and completed by the participants, other negative implications were expected to ensue. For example, the true effect of the Catalogue-assisted treatment of the independent variable in the real experimental exercise, was likely to be confounded by the additional experience that the participants were expected to gain as a result of completing the pre-test task(s). Because of these problems, it was decided that questionnaires were a better option. Due to the impossibility and impracticality of collecting information about all possible non-treatment related factors, it was decided to collect information about the software development background of the participants while assuming that the remaining factors were at a fixed level for all participants (see further section 4.2.5.1). With the adoption of option two, figure 4.2 can be evolved into figure 4.3.

![Figure 4.3](image_url)
Assigning Groups to Treatments of the Independent Variable

In order to simplify the referencing to the various groups involved in the field experiments, three definitions were made:

i) the participants taking the unit in the Mount Lawley campus during semester 1, 2001, are hereinafter referred to as the ML1 group.

ii) the participants taking the unit in the Joondalup campus during semester 1, 2001, are hereinafter referred to as the JO1 group.

iii) the participants taking the unit in the Mount Lawley campus during semester 2, 2001, are hereinafter referred to as the ML2 group.

Group size was used as a criterion to assign the groups to the treatments. Table 4.7 shows the arrangements made.

<table>
<thead>
<tr>
<th>Semester</th>
<th>Treatment</th>
<th>ML1 (N=133 participants)</th>
<th>JO1 (N=39 participants)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semester 1 2001</td>
<td>Catalogue-assisted Java Development</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Semester 2 2001</td>
<td>Standard Java Development</td>
<td>ML2 (n=67 participants)</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.7 suggests that ML1 and ML2 participants would use a Catalogue of Errors in their development, whereas the JO1 participants would not. The Catalogue of Errors is the intervention into the standard development. Consequently, the ML1 and ML2 groups are the intervention groups, whereas the JO1 group is the non-intervention group.

Table 4.7 also shows that both treatments of the independent variable were run in semester 1, 2001, but only one treatment (the Catalogue-assisted Java development) of the independent variable was run during semester 2, 2001. The decision was purely opportunistic in nature. Since the unit was only offered at one campus, any segregation of the ML2 group of participants into intervention and non-intervention groups would pose the data sharing problems.

The reason for running the Catalogue-assisted Java development treatment a second time was to conform with the need for replication requirement that most experimental publications necessitate (Basili et al., 1996; Brooks, Daly, Miller, Roper, & Wood, 1994; Judd et al., 1991; Shull et al., 1999; Shull, 1998). If the results of a replication are
consistent with the original research, not only is the original research validated, but also confidence in the previous results is enhanced. While exact replications are often impractical and unattainable due to cost considerations and the need for original work, partial replications or replications with extensions can be used (Brooks et al., 1994; Hubbard, Vetter, & Little, 1998). Also:

"Replication takes two forms: internal and external. Internal replication is undertaken by the original experimenters; external replication is undertaken by independent researchers and is critical for establishing sound results." (Brooks et al., 1994, p. 2)

The field experiment conducted during semester 2, 2001, was therefore an internal partial replication of the field experiment conducted in semester 1, 2001. In semester 1, 2001, the ML1 and JO1 participants were required to develop the Intersection Simulator. In semester 2, 2001, the ML2 participants were required to develop the Hotel Lobby Simulator (see table 4.2). While the Intersection and the Hotel Lobby Simulators were different, they were of similar complexity (see appendix B, section 2.1.3). The results obtained during semester 2, 2001, (with group ML2) would be examined for consistency with those of the Catalogue-assisted Java development treatment run during the semester 1, 2001 (with group ML1). If the findings of both experiments are consistent they have more confirmatory power. Even greater confirmatory power can be obtained by future external replications, which can be a future research direction. This would enhance the generalisability of the results (Brooks et al., 1994).

4.2.4.4 Balancing

Balancing involves assigning an equal number of participants to each treatment of the independent variable (Fenton & Pfleeger, 1997; Wohlin et al., 2000). As suggested in the previous section the replication of the field experiment in Semester 2, 2001, did not involve a control group. Consequently, such internal partial replication is not affected by balancing. In the following discussion, balancing is only discussed with reference to the initial trial of the field experiment in Semester 1, 2001.

Appendix B, section 2.1.3 compares and contrasts between the Intersection and the Hotel Lobby Simulators.
According to Wohlin et al. (2000), balancing is desirable because it helps strengthen the statistical analysis. However, balancing is not necessary (Wohlin et al., 2000). Fenton & Pfleeger (1997) agree with Wohlin et al. (2000) by suggesting that experimental design can range from being completely balanced to having little or no balance at all. Miller et al. (1997) are more cautious than Wohlin et al. (1997) and Fenton & Pfleeger (1997) when balancing is considered, for example:

"Once the required sample size has been calculated and the subjects [participants] recruited it is important to separate into groups of approximately equal numbers, N, say. If this is not exercised, the skewed distribution of subjects results in a lower power of statistical test because a subset of subjects will contribute nothing to the study." (p. 289)

It is clear that in the field experiment that took place in Semester 1, 2001, the assignment of the treatments of the independent variable to the groups (N\text{ML1}=133; N\text{J01}=39) suggests an unbalanced design. To help determine the number of participants whose data run the risk of not being fully utilized in unbalanced designs, and to determine the effect that this may have on the power of the statistical analysis of an experiment, Miller et al. (1997) advocate the computation of what they refer to as the harmonic mean. The harmonic mean is computed by the following formula:

\[
N = \frac{2 \times N_1 \times N_2}{(N_1 + N_2)},
\]

where \(N_1\) and \(N_2\) refer to the size of groups.

The harmonic mean for the field experiment is 60\(^{16}\). Therefore, the field experiment with a harmonic mean of 60, is equivalent to one with equal group size of 60 rather than 86\(^{17}\) (Miller et al., 1997). This suggests that the skewed distribution of the participants in ML1 and J01 fails to fully utilize 52\(^{18}\) participants' data (Miller et al., 1997). Clearly, the distribution of the participants into the ML1 and J01 groups such that \(N_{\text{ML1}}=133\) and \(N_{\text{J01}}=39\) will not maximize the power of the statistical analysis. However, as discussed above, the samples were convenience samples and the investigator was unable to ask participants to move to a different group. This situation is not uncommon in software engineering research. In this context, Miller et al. (1997) state that:

\(^{16}\) \(\frac{2 \times 133 \times 39}{(133 + 39)} = 60.\)
\(^{17}\) \((133 + 39)/2 = 86.\)
\(^{18}\) \((133 + 39) - (60 + 60)) = 52.\)
"Admittedly, however, it is not always possible to split subjects [participants] into even groups. In such cases it is always better to use the 'extra' subjects rather than simply discard them." (p. 289)

Therefore, the issue is whether the power of the statistical analysis is that inherent with the existing arrangement, where \( N_{ML} = 133 \) and \( N_{101} = 39 \), affects the reliability of the results. This is discussed in detail in section 5.1.1 (see Chapter five), which deals with the low statistical power threat to the field experiment.

4.2.5 Instrumentation

The objective of this section is to present the various types of instruments that were used in the field studies and experiments. These have been classified into three major categories, namely,

i) Data collection instruments;

ii) Training instruments; and

iii) Development instruments.

Some of the instruments included in these categories are based on the study reported in (Basili et al., 1995). These instruments are presented in appendix B.

4.2.5.1 Data Collection Instruments

The design of data collection instruments constitutes a compromise among conflicting objectives (Basili & Weiss, 1984). Obviously, it is desirable to obtain as complete and detailed a set of data as possible in order to address in depth the research questions of interest. However, this entails designing data collection instruments that are large and detailed, and which require time and effort to complete. Data collection instruments of this nature are likely to irritate participants (Basili & Weiss, 1984). Consequently, the design of the data collection instruments was governed by a set of criteria adapted from Basili & Weiss, (1984) and Littlefair, (2001) which are synthesized below:

i) The data collection instrument should be as straightforward as possible;

ii) The collected data should be readily and easily transferable to statistical packages to facilitate processing;

iii) The individual participants should require only a minimal amount of time to complete the data collection instruments;
iv) The data collection instruments should permit participants some flexibility to describe any change or personal opinions outside the scope of the questions asked.

v) Overall participant data entry overhead should be minimal and avoided where possible.

Two classes of data are used in this research, namely, *participant-generated data* and *reviewer-generated data*. Participant-generated data were collected via questionnaires, which were used to seek feedback from participants regarding various issues. Reviewer-generated data were generated by four reviewers who reviewed the participants' development artifacts. The rationale for gathering data and the individual data collection instruments are presented in the following sections.

**Participant-generated Data**

The data collection instrument that was used to collect data from participants was the questionnaire. The primary reason for using questionnaires was that they constitute the best way to collect information from participants in a timely and efficient manner. As indicated previously, there were 239 participants, 198 of whom underwent the Catalogue-assisted development treatment and were expected to provide feedback concerning the Error Framework and the Catalogue of Errors. Had individual interviews been used, due to practical reasons, they would have had to take place some time after the event, which would have resulted in participants having to rely on their past memory to retrieve the required data. Given the size of the sample, it was decided that such an approach, while extremely time-consuming, was also more likely to lead to incomplete and inaccurate data. Participants can be asked to fill in questionnaires immediately after the event, while the information is still fresh in their minds. An explanation about the individual questionnaires is provided in the following sections.

**Software Development Background Questionnaire**

The objective of this questionnaire was to collect data on the comparability of the software development background of the participants. This would also enable a better understanding of the results obtained in this research. A copy of this questionnaire has been provided in appendix B, section 2.2.1.1. The analysis of the data that was collected
via the Software Development Background Questionnaire has been presented later in this chapter (see section 4.4.4).

**Error Framework and Catalogue of Errors Evaluation Questionnaires**

Both the Error Framework and the Catalogue of Errors Evaluation Questionnaires contain assertions which participants were expected to evaluate on a five-point scale. The choice to use five point scale evaluation as part of the questionnaires was motivated by three factors:

i) The use of an odd number of options provides the possibility for inclusion of a neutral option as the middle point of the scale that reduces the potential to bias the evaluation. This would be better than alternatives which either do not provide a neutral option or provide options counting a different numbers above and below the neutral point in the scale.

ii) A 3-point scale could have been used, however, it was feared that it would have the potential of missing out meaningful and useful information.

iii) Finally, using an odd number of greater than five points on the scale was regarded as a possible way to increase the variation of the collected data, without adding considerable value to the overall results.

The objective of the Error Framework Evaluation Questionnaire is to collect the feedback from participants about the various perspectives of the Error Framework. The participants were expected to evaluate the perspectives of the Error Framework on a 5-point usefulness scale. This feedback would not only demonstrate whether the participants recognized the various perspectives of the Error Framework, but would also obtain their empirical assessment of the usefulness of the knowledge represented by such perspectives. This was important because it helps address the *usefulness* success factor that was identified in chapter three, section 3.4.1.

The objective of the Catalogue of Errors Evaluation Questionnaire was to gather feedback from participants in order to evaluate some of research question two success factors (see section 3.4.2, chapter three) including:

i) the implementation of Error Framework perspectives in the Catalogue of Errors;

ii) the usability of the Catalogue of Errors;

iii) the usefulness of Catalogue of Errors to identify and correct errors; and
iv) the overall software development skills improvement.

The learnability of the Catalogue of Errors success factor was addressed by collecting Catalogue of Errors learning time data.

As will be shown in chapter seven, the Catalogue of Errors was divided into three components, namely Catalogue of Requirements, Design, and Code errors. In this context, three Evaluation Questionnaires were designed addressing the same issues of each component of the Catalogue of Errors. For example, one of the aspects of the Error Framework is called error cause. The assertion to evaluate error cause implementation in the Catalogue of Requirements, Design, and Code errors is similar to the following:

"The Catalogue of (Requirements/Design/Code) Errors has helped me understand the causes of the errors included in it."

Similar assertions were made about how the other perspectives of the Error Framework were implemented in the Catalogue. Assertions were also made about the various success factors of research question two. The participants were expected to evaluate such assertions on a 5-point agreement-disagreement scale. The participant evaluation of the Catalogue of Errors was not only considered as the means to empirically evaluate the Catalogue, but also as an opportunity to gather clues to further enhance the Catalogue.

Both the Error Framework and the Catalogue of Errors Evaluation Questionnaires allowed participants to provide open-ended feedback about any aspect of the Error Framework or Catalogue of Errors. This was done in order to enable the participants to provide comments outside the questions and to openly criticize the Error Framework and the Catalogue of Errors.

A detailed discussion about the feedback of the participants with regard to the Error Framework and Catalogue of Errors is presented in chapter six and seven, respectively. Chapter six and seven also discuss how the data representing participant feedback is used to address research question one and two success factors. The Error Framework
Evaluation Questionnaire and the Catalogue of Errors Evaluation Questionnaires can be found in appendix B, section 2.2.1.2 and 2.2.1.3.

**Artifact Development and Catalogue Learning Time**

Participants were also asked to indicate the amount of time that they had spent on developing the different software development artifacts as well as the amount of time that they had spent on learning the Catalogue of Errors. The artifact development time is important because it would help address the productivity success factor for research question three (see section 3.4.3, chapter three). The time spent to learn the Catalogue of Errors is important because it helps address the Catalogue of Error Learnability success factor for research question two (see section 3.4.2, chapter three).

The requirement to indicate the time spent to complete an artifact in general or a given task in particular is common for programming units in the university where the research took place. Such a requirement occurs in the context of the Intellectual Property requirement, where a student is expected to show a timed log of their activities to complete the development artifact(s). Given such an arrangement, it was decided not to require participants to provide artifact development time data separately via a questionnaire, when the participants would provide such information, anyway.

**Reviewer-generated Data**

As suggested in table 4.2, individual participants undergoing both treatments of the independent variable would produce development artifacts at the completion of various development phases. In order to address question three and some of its success factors (see section 3.4.3), data about the number and types of errors committed by all participants and the size of the different development artifacts were collected. Error type/count data and artifact size data issues are addressed in the following sections.

**Error Type and Count Data**

The error type\(^{10}\) and count data are important because they help address the success factors for question three. The development artifacts where error type and count data were located were also deliverables, which the student participants were supposed to

---

\(^{10}\) The reviewers examined the three artifacts submitted by the participants for errors that were catalogued in the Catalogue of Errors and errors that were not.
submit in order for them to be assessed for the Internet and Java Programming unit. Error count and type information identified in a development artifact was provided in terms of comments and returned to the participants so that the comments could be used to improve future development artifacts (see appendix D). Error count and type information was also later recorded in Error Report Forms (see appendix B, section 2.2.2.1).

The fact that the error type and count data collection effort was absorbed by the reviewers is considered advantageous because it can be a “source of irritation” (Basili and Weiss, 1984, p.432) to the participants.

The possibility of getting the participants themselves to turn in a list of errors committed in their development artifacts, as done by Shull et al., (1999) was looked into and immediately discarded for three main reasons:

i) The consistency with which errors would be discovered and reported would be very difficult to maintain due to the large number of participants involved;

ii) The error lists turned in by the individual participant could have been biased, given that the artifacts were also deliverables requiring assessment in partial completion for the unit where the participants were enrolled;

iii) The issue of the identification of false positives would have been serious, given the large number of participants.

Another advantage associated with the decision to get reviewers to collect error count and type information is the increased consistency with which errors would be reported due to the fact that a minimal number of people would be involved.

However, despite the advantages of using reviewers to collect error type and count data, it was recognized that there was a potential pitfall that could threaten the quality of the data: the likelihood of bias from the reviewers. This was recognized early before error counts and type data were actually collected and was addressed by developing and using standardized development Artifacts Review Guides, summaries of which are shown in appendix B, section 2.2.2.2. In addition, brief informal pre- and post-review meetings were held between reviewers. The pre-review meetings focused on what errors to identify and how to identify them and report them back to the participants. This was
carried out to ensure that all reviewers started their reviews with consistent error definitions in mind. The post-review meetings focused on the consistency with which errors were identified in the development artifacts and subsequently reported. In order to ensure this further, randomly selected samples of four reviewed development artifacts were frequently exchanged between some reviewers. Pre- and post-review meetings were held for each development artifact. In the study reported in Basili et al., (1995) error counts and type data were collected in a similar manner.

**Artifact Size Data**

Artifact size data are important because they help address some of the success factors of research question three (see section 3.4.3, chapter three). Two types of artifacts were produced by the participants: code and non-code-based artifacts. Code-based artifacts contain code written in the Java programming language, whereas non-code-based artifacts contain text describing requirements or design issues of the simulator projects. The size of code-based artifacts was determined by the number of non-commentary Lines of Code, which was measured using the JavaNCSS (version 15.32) (Lahme, 2001). The size of non-code-based artifacts was determined by counting the number of pages contained in the artifact. To ensure consistency between the artifacts developed by the participants, the participants were required to adopt the same Artifact Templates (i.e. artifact format) to present their code-based and non-code based artifacts (see appendix B, section 2.1.4).

**4.2.5.2 Training Instruments**

The objective of this section is to describe the training instruments that were used to train the participants who underwent the Catalogue-assisted Java development treatment. The training instruments are divided into two categories, namely, Error Framework training instruments and Catalogue of Errors training instruments. Both categories of instruments were documented in lecture notes and handouts. The lecture notes constituted the summarized version of the information provided in the handouts. Lecture notes and handouts were prepared for three main reasons:

i) To facilitate the teaching and learning of the Error Framework and the Catalogue of Errors; and

ii) To be consistent with the teaching and learning standards of the university where the research took place.
The study of the Error Framework and the Catalogue of Errors was incorporated into the objectives of the Internet and Java Programming unit. Therefore, the materials had to be made accessible to the students.

The lecture notes can be found in appendix B, sections 2.3.1 and 2.3.2, whereas the handouts have been included in the appendix A, section 1.2.

4.2.5.3 Development Instruments
The development instruments are comprised of the simulator applications that the participants were expected to develop. As indicated earlier, two different simulator projects were developed, namely, the Intersection Simulator and the Hotel Lobby Simulator. The Intersection Simulator was developed by the participants in semester 1, 2001. A second simulator was developed in semester 2, 2001, to ensure that the participants in semester 2 did not adopt their seniors' simulators as their own. Both simulators involved the development of requirements, design and code artifacts. The simulators, however, while different, were of a similar complexity. The problem statements describing the simulators and an account of their similarities have been included in the appendix B, sections 2.1.1, 2.1.2, and 2.1.3.

4.2.5.4 Summary
The diagram presented in figure 4.4 summarizes the instruments that were used to capture the different types of data needed. Note that the Density of Error Type, and Developer Productivity and Escape Ratios are computed using data that is provided directly either by the reviewers or by the participants or both. Note also that the shaded parallelograms and the trapezoid group together the instruments used to collect data concerning the research questions discussed in chapter three. Table 4.8 summarizes the research questions, the success factors as well as the data that are required to address each success factor.
Figure 4.4 - Summary: Instruments and Data
4.3 Research Operation Issues

This section describes the operational phase of the research. The operational phase requires two main issues to be addressed. Firstly, the participants and instruments must be prepared. Secondly, the collected data must be validated.

4.3.1 Preparation

Before the plan is executed, the participants need to be prepared for participation and any instrumentation concerns must be properly addressed. Such preparation will ensure the smooth execution of the research.

4.3.1.1 Committing the Participants

The objective of this section is to address issues related to the commitment of participants. Specifically, the issues that are addressed below include disclosing the research objectives, obtaining consent from the participants, addressing the sensitivity of the results, providing inducements to participants, and other ethical considerations. These issues are addressed in turn in the following sections.

## Table 4.8 – Summary: Research Questions, Success Factors and Data

<table>
<thead>
<tr>
<th>Research Questions and Success Factors</th>
<th>Addressing the Success Factors</th>
<th>Addressed In</th>
</tr>
</thead>
<tbody>
<tr>
<td>Orthogonality</td>
<td>By Investigator</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Comprehensiveness</td>
<td>By Investigator</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Useability</td>
<td>By Investigator</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Usefulness</td>
<td>5-point scale Usefulness Evaluation via Error Framework Evaluation Questionnaire</td>
<td>Chapter 6</td>
</tr>
<tr>
<td>Useability</td>
<td>Catalogue of Errors Learning Time</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Implementation of Error</td>
<td>5-point scale Agreement via Catalogue of Errors Evaluation Questionnaire</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Useability</td>
<td>5-point scale Agreement via Catalogue of Errors Evaluation Questionnaire</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Usefulness to Identify and Correct Errors</td>
<td>5-point scale Agreement via Catalogue of Errors Evaluation Questionnaire</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Overal Development Skills Improvement</td>
<td>5-point scale Agreement via Catalogue of Errors Evaluation Questionnaire</td>
<td>Chapter 7</td>
</tr>
<tr>
<td>Error Density</td>
<td>Density of Error Type (Error Type/Count and Size of Artifact)</td>
<td>Chapter 8</td>
</tr>
<tr>
<td>Escape Rates</td>
<td>Error type/cont</td>
<td>Chapter 8</td>
</tr>
<tr>
<td>Productivity</td>
<td>Developer Productivity (Time Spent to Develop an Artifact and Size of Artifact)</td>
<td>Chapter 8</td>
</tr>
</tbody>
</table>
Disclosing Research Objectives

A document called Statement of Disclosure was distributed to all participants before the start of the field studies and field experiment (AVCC, 1997). The document contains information and outlines the objectives of the research. The potential benefits and burdens associated with the experiment and the direct or indirect ways in which they could affect the participants were also explained (Zikmund, 1994). In order to avoid adverse effects on results, the hypotheses and information about the data to be collected, were not explicitly disclosed to the participants (Fenton & Pfleeger, 1997; Host et al., 2000; Wohlin et al., 2000). From the participants' point of view, they were not participating in a research exercise; but were taking a unit in partial fulfilment of their degrees and completing the required assignments (i.e. the development artifacts) in order to fulfil the unit objectives. The Statement of Disclosure has been provided in appendix B, section 2.4.1.

Obtaining Participant Consent

In order to comply with ethics rules and regulations set out by Edith Cowan University Committee for the Conduct of Ethical Research and to maximise the validity of the collected data, participants' consent was sought so that the investigator would be able to collect and use the data. After the contents of the Statement of Disclosure was explained to the participants, consent forms were handed out for signing. This constituted a formal agreement allowing the investigator to collect and use the data concerning the participants' work. However, it was made clear that the participants were free not to sign the consent forms or to withdraw their consent at anytime, if they chose to do so, without having to provide any explanation for their choice. The consent was obtained on an individual basis using the relevant pre-defined forms provided by the Committee for the Conduct of Ethical Research, Edith Cowan University (see appendix B, section 2.4.2). The number of the participants who signed the consent forms were shown earlier in the chapter (see table 4.3).

Sensitivity of the Results

The data collected related to the participants' ability to produce artifacts without committing errors as well as their productivity. In this context, confidentiality is an important concern. The participants were guaranteed anonymity and the collected data was to be immediately de-identified to avoid direct reference to individual participants.
The collected data was stored in designated locations as approved by the Committee for the Conduct of Ethical Research (Australia, 1999; Crothers, 2000).

**Inducements**

Incentives and motivation are necessary so the participants can provide their best performance. One positive inducement was the expectation that the Error Framework and the Catalogue of Errors documentation and training would help participants learn more and improve their development skills through knowledge of errors. Secondly, given that the Error Framework and the Catalogue of Errors were part of the Internet and Java Programming unit objectives, it was indicated upfront that these subject areas would be assessable.

One negative motivational factor was that some participants could regard the Error Framework and the Catalogue of Errors documentation and training as additional work that was not included in the unit in the previous semesters. A few participants expressed such concerns. These concerns were addressed by the investigator, lecturer and/or unit coordinator who reiterated the fact that the Error Framework and Catalogue of Errors had been incorporated into the unit objectives and hence were part of the unit. This was reinforced continually both in formal and informal meetings with the students.

**Other Ethical Considerations**

Another ethical issue is concerned with the assurance that no group is unfairly discriminated against. This must be achieved by providing all groups involved in the research with an equal opportunity to acquire knowledge (Australia, 1999; Crothers, 2000). The design of the research approach described earlier in this chapter had the potential for the discrimination of the JO1 group as opposed to ML1 and ML2 to occur (see table 4.7). While ML1 and ML2 would be provided with documentation of and trained with the Error Framework and Catalogue of Errors, the JO1 group would not receive such documentation and training during the semester. To address this ethical issue the following arrangement was made:

1. The JO1 group received no Error Framework and Catalogue of Errors training for the duration of the project development (12 weeks).
2. The JO1 group received complete and extensive Error Framework and Catalogue of Errors training and documentation during week 13 of the semester.
3. The ML1 and ML2 groups received Error Framework and Catalogue of Error training and documentation for the duration of the project (12 weeks).

4. The results of each participant's project were to be used to assess the in-semester work component of the assessment for the unit where the participants/students were enrolled. Such results would, therefore, be subjected to a moderation exercise, if necessary.

4.3.1.2 Instrumentation Concerns

Individual sets of instruments including training materials and questionnaires, etc. were distributed to each participant. The participants were also briefed about how the questionnaires were to be filled in. Participants were expected to keep the training materials and hand the questionnaires back to the investigator. Initially, they were required to supply their student ID number as a means of identification. There were two reasons for this request:

i) Five questionnaires (i.e. Software Development Background Questionnaire, Error Framework Evaluation Questionnaire, and Catalogue of Errors Evaluation Questionnaire for the three components of the Catalogue of Errors) were required to be filled in by each participant at different times. The student ID numbers would be used to put the data for each participant together into spreadsheets.

ii) It was recognized that sometimes questionnaires could be filled in an unclear or conflicting way. When this occurred, the student ID number was needed to trace the participant(s) in order to obtain clarifications. Once the questionnaires were validated and found to be clearly filled in, they were de-identified to make participant identification impossible.

4.3.2 Data Validation

The objective of this section is to describe data collection and validation issues. Basili & Weiss (1984) suggest that:

"Validation consists of checking the questionnaires for correctness, consistency, and completeness." (p.730).

They also propose a number of issues that need consideration as part of data collection and validation. These are discussed in the following sections.
Misunderstanding of the data collection procedures

This deals with whether the data collection procedures were properly understood by the reviewers and the participants. In order to minimise any misunderstandings each time the participants were expected to fill in a questionnaire, a briefing session was conducted to explain the questions and how to answer them. Second, the reviewers of the artifacts produced by the participants were also shown how to identify errors of various types and report them on the artifacts as feedback to students. This was done in pre-review meetings.

Timely data collection and validation

The shorter the lag between data collection and validation the better. Data collection and validation should also be concurrent with software development. This was applied in two contexts. In this study, the reviewers validated the data within fourteen days after the participants turned in their artifacts. This is considered to be reasonably concurrent with the development of the artifacts. The questionnaires were validated for accuracy immediately after they were submitted. Generally, questionnaire submission occurred at the same time as the artifact submission. When submitted, the questionnaires were examined. In general, the following flaws were observed:

i) Ticking between the entries (rather than on the entry itself) in the 5-point scale evaluation questionnaires. This problem was rectified by approaching the participants that had committed this mistake and asking them to fill in the questionnaire(s) again.

ii) Failure to indicate student ID numbers enabling the investigator to bundle individual participant data. In some instances this was easy to rectify, because the participant(s) had included the unidentified questionnaire with the artifact, which was easily identifiable. To maintain anonymity artefacts and questionnaires were immediately separated. In 13 cases the questionnaires were submitted anonymously. Calls were made for these participants to resubmit the questionnaires and supply their ID numbers. However, the data supplied via those questionnaires was eventually disregarded, because none of the 13 participants came forward.
**Sensitivity of collected data**

According to Basili & Weiss (1984), the participants need to be assured that the data they would provide via the questionnaires, was not to be used to their disadvantage. Participants were, therefore, reassured that the data collected from their artifacts and questionnaires would not be used to bias their grading for the unit.

**4.4 Data Analysis Methods and Presentation Issues**

**4.4.1 Overview**

The objective of this section is to present an overview of the methods that were used to analyze and present the collected data. As shown in figure 4.4, the collected data seek to characterize various attributes of development artifacts. An attribute constitutes a characteristic of an artifact that an investigator is interested in (e.g., size of the code artifact in terms of LOC, or an evaluation of the Error Framework, etc.). The characterization of an attribute with data is known as measurement of the attribute. Measurement can occur by using different scales. A measurement scale is a system that dictates how to map between the attribute one is seeking to characterize and a predefined set of recognizable and meaningful numbers or symbols which represent a summary of all possible characterizations of that attribute. The definition of measurement scales and the determination of the scale that is used to measure specific attributes is important, because it can determine the kind of data analysis method to use (De Vaus, 2002; Fenton & Pfleeger, 1997; Littlefair, 2001). Fenton & Pfleeger (1997) classify measurement scales as one of the following five types:

i) The nominal scale is the least powerful and the most primitive scale which seeks to define unordered classes of categories and place the attribute under consideration in one of them (e.g. error typing);

ii) The ordinal scale augments the nominal scale by adding information about the ordering of the underlying classes, thereby allowing them to be ranked (e.g. software complexity).

iii) The interval scale preserves the order introduced by the ordinal scale and supplements the notion of the size or magnitude of the intervals between the classes (e.g. time measured in minutes, hours, days, months etc. for a software project to complete).
iv) The ratio scale retains ordering, the size of the intervals and augments a zero element, which represents lack of presence of the attribute and the initial point where the other classes start increasing at equal intervals. Arithmetic and ratios, including division and multiplication, are also supported (e.g. length of software code in LOC).

v) The absolute scale indicates the number of occurrences of a certain entity, namely the actual count. This type of scaling supports all arithmetic operations, including division and multiplication (number of failures, number of errors, etc.).

It is clear that the sets of ordinal, interval, ratio and absolute scales are successive supersets of the preceding sets of scales (Littlefair, 2001). An extensive description of the types of measurement scales can be found in Conte et al., (1986); Fenton & Pfleeger, (1997).

The data collected for research question one (Error Framework Evaluation Questionnaire: 5-point perspective usefulness evaluation and open-ended evaluation) were similar in nature to the data collected for question two (Catalogue of Errors Evaluation Questionnaire: 5-point scale agreement and open-ended evaluation) (see figure 4.4). The 5-point scale evaluations for both research question one and two were therefore measured using the ordinal scale. The open-ended evaluations are subjective evaluations about either the Error Framework or the Catalogue of Errors. Consequently, the data analysis methods for both research questions one and two are similar, and are therefore presented together in section 4.4.2. The data collected for research question three were measured by using at least the interval scale (see figure 4.4). Section 4.4.3 presents the method by which question three data were analyzed.

4.4.2 Research Question One and Two Data Analysis and Presentation
The ordinal data collected to answer question one and two are mainly presented in a tabular format. The tables contain the median values and the interquartile ranges of the underlying distributions. The median is used to measure the central tendency of a distribution of ordinal data and constitutes the midpoint in the distribution (of participant indications in the 5-point scales of the evaluation questionnaires) after the individual distribution values have been ordered from the smallest to the largest (Mason
There are as many values above the median as there are below it. The variation of ordinal distributions is shown using the interquartile range measurement, which constitutes the middle 50 percent of the distribution after the bottom and the top 25 percent (i.e. 25th and 75th percentiles) have been dropped. Narrow interquartile ranges show a lower variation and as a consequence the median is a better representative of the underlying distribution (De Vaus, 2002). De Vaus (2002) suggests that the median and the interquartile range are the best statistics that are suitable to describe data of an ordinal scale.

The subjective open-ended evaluations were examined individually for all participants who chose to provide them. These evaluations were abstracted and summarised in tables where the frequencies of the participants subscribing to a given evaluation were included. Both the descriptive statistics (i.e. median and interquartile range) and the frequencies of the abstracted open-ended evaluations were categorised on the basis of group (i.e. ML1, JO1, and ML2). The data analysis and presentation presented above can be found in chapter six and seven, which deal with research questions one and two.

4.4.3 Research Question Three Data Analysis and Presentation

The data collected to address question three are measurements of an interval scale. According to De Vaus (2002) interval scale distributions are described in terms of means and standard deviations. Means and standard deviations are organised in a tabular format and are used to describe the central tendency and the variation of the underlying distributions. The mean of a distribution is the ratio between the sum of the individual values of the distribution and the total number of values in the distribution (Keller, Warrack, & Bartel, 1994). The standard deviation describes how far away the individual values of the distribution are from the mean (Iversen & Gergen, 1997). The descriptive statistics (i.e. mean and standard deviation) were categorised on the basis of group (i.e. ML1, JO1, and ML2). In order to aid the descriptive power of the above statistics, box plots were also used (De Vaus, 2002; Keller et al., 1994).

The data collected for question three were also analysed using inferential statistics. As shown in section 4.2.4.2, the research hypotheses are expressed in terms of pairs of statistical hypothesis (null hypothesis (H0) versus alternative hypothesis (H1)). A hypothesis must be tested using statistical tests. While the alternative hypothesis (H1)
presupposes a significant difference between the treatments of the independent variable, the null hypothesis \( (H_0) \) assumes no such difference. Statistical significance tests are expected to show rejection or non-rejection of \( H_0 \) as opposed to rejection or non-rejection of \( H_1 \). De Vaus (2002) describes the objective of statistical tests of significance as follows:

"... tests of significance tell us how likely it is that the sample findings would exist within the population as a whole. To be precise, tests of significance tell us how likely it is (with a given sample size) that we would find the particular pattern observed in a sample if such a pattern did not really exist in the population. That is, how likely is it that the sample figures do not reflect population figures." (p. 208).

This is the reason why tests of significance are also known by the name of inferential statistics, because they help the investigator infer about the likelihood that observed sample patterns exist in the population where the sample is drawn from (De Vaus, 2002). There are many statistical tests available in literature that can be used to achieve this (Keller et al., 1994; Mendenhall & Sincich, 1992; Walpole & Myers, 1993; Wohlin et al., 2000).

The choice of a statistical test depends on whether the collected data satisfies certain required assumptions. For example, the so-called parametric tests are used to compare two groups when the data collected is normally distributed, whereas non-parametric tests are used when the normality assumption is violated.

In order to determine whether the data collected in the field experiments was normally distributed, the Kolmogorov-Smirnov test was used. However, this test revealed that the normality assumption for the collected data was invalid (see appendix C, sections 3.3.1, 3.3.2, and 3.3.3), which prompted the need to use non-parametric statistical tests to test the hypotheses. Therefore, the Mann-Whitney U non-parametric statistical test was used. This statistical test is based on the ranking of distribution values in two groups and testing whether the distributions are equivalent in locations (Huck, 2000; Land, 2000). The Mann-Whitney U test is normally used when distributions from two independent samples are compared. This test was used to compare the results between the ML1 and JO1 participants. The results of the statistical tests for the formulated
hypotheses are presented in tables that contain the dependent variable with reference to each statistical test. Tables also include the sample size (i.e. N) and the computed probability (i.e. p value) at the 95% confidence interval. The data analysis and presentation issues presented above can be found in chapter eight, which deals with research question three.

4.4.4 Software Development Background Questionnaire Data Analysis

The groups that participated in the field experiments were not significantly different from each other with respect to their software development background. The data collected via the Software Development Background Questionnaire violated the normality assumption (see appendix C, section 3.1.1). Consequently non-parametric statistical tests were used in order to compare the three groups (i.e. ML1, JO1, and ML2). The Kruskal-Wallis H test was identified as a suitable test for this purpose. This test is based on the ranking of distribution values of three or more groups, and like the Mann-Whitney U test, it tests whether the distributions are equivalent in locations (Huck, 2000; Land, 2000). The difference is that while the Mann-Whitney U test is used when distributions from two independent samples are compared, the Kruskal-Wallis H test is used when distributions from three or more samples are compared (Huck, 2000).

The results of the Kruskal-Wallis H test indicate that the software development backgrounds of the ML1, JO1, and ML2 participants, as characterised by the data collected through the Software Development Background Questionnaire, are not significantly different (see appendix C, section 3.1.2).

4.5 Summary

This chapter has examined issues relating to the design of the research approach. Initially, planning issues related to the research approach were covered. Planning issues include context and participant selection, field experiment design, instrumentation, and data analysis and presentation.

It was argued and concluded that a university environment constitutes a good context and that student participants constitute an attractive participant selection pool in order to investigate the three research questions. Convenience sampling was used to recognise
three groups of participants, namely, ML1, JO1, and ML2. ML1 and JO1 participated in
the field studies and the field experiment during Semester 1, 2001, whereas ML2
participants were used during Semester 2, 2001.

Due to the convenience considerations in selecting student participants, the field
experiments lacked randomisation. Consequently, they had to be treated as quasi-
experiments. The review of literature indicates that when quasi-experimental settings
are used the additional task of establishing comparability between ML1, JO1, and ML2
participants must be carried out. This was done by collecting additional information
from the participants, which allowed the investigator to conclude that any changes in the
values of the dependent variable were due to the effects of the treatments of the
independent variables.

Instrumentation issues as well as a detailed account of the instruments used in the field
studies and the field experiments were also discussed in this chapter. Three broad
categories of instruments were recognised, namely, data collection instruments, training
instruments, and development instruments. The discussion on instrumentation showed
how individual instruments would contribute towards addressing the three research
questions.

Issues relating to context and participant selection and instrumentation concern all of the
research questions, whereas the field experiment design issues concern only research
question three.

Chapter four also dealt with issues concerning research operation, including participant
preparation and data validation. These issues were important because the participants
had to be adequately prepared before embarking into the field studies and the field
experiments. Also the data collected from the field studies and the field experiment to
address the three research questions had to be properly validated before analysis.

Finally, the chapter was concluded with a discussion of how the collected data would be
analysed and presented in order to address the research questions.
5. Validity Evaluation

5.1 Overview

In chapter four it was shown that two quasi-experiments were conducted in order to confirm or refute propositions about the ability of developers to commit and remove errors and their productivity with and without a Catalogue of Errors. These propositions were articulated in terms of hypotheses which establish relationships between dependent and independent variables. Unfortunately, both the field experiments and quasi-experiments generate data only from samples in order to make conclusions about the relationships between the dependent and independent variables. Attempts are then made to infer such conclusions to the population where the samples were drawn from. This occurs because, due to practical and feasibility considerations, it is impossible to involve the entire population into experimental exercises. Consequently, in any experiment or quasi-experiment, strictly speaking, it is impossible to know the truth or the falsity of any propositions made about the true relationship between the dependent and the independent variables for the entire population (Cook & Campbell, 1979; DeVaus, 2002). Therefore, the term validity refers to the best possible approximation of the relationships between the dependent and the independent variables for the participants of the study, to the true relationships between the dependent and the independent variables for the entire population of Java student developers (Cook & Campbell, 1979; Frankfort-Nachmias & Nachmias, 1992; Judd et al., 1991). It is, therefore, important that henceforth the term validity be understood to represent approximate or tentative validity.

Cook & Campbell’s (1979) landmark work on quasi-experimentation suggests that, in order to assess the validity of an experiment, four questions need to be answered:

i) Is there a relationship between the dependent and the independent variables defined in the experiment?

ii) If a relationship between the dependent and the independent variables is ascertained, is such a relationship plausibly causal from the independent variable(s) to the dependent ones?

iii) If a relationship between the dependent and the independent variables is ascertained, and if it is reasonably known that it is the independent variable(s)
causing the effect on the dependent variables, what particular cause and effect constructs are associated with such a relationship?

iv) How generalisable is the causal relationship between the independent variable(s) and the dependent variables to other individuals of the target population who may be operating in other settings, times, etc.?

According to Cook & Campbell (1979), Judd, Smith, & Kidder (1991), Frankfort-Nachmias & Nachmias (1992), and Wohlin et al. (2000), each of the above questions involves numerous issues that need to be addressed. Cook & Campbell (1979) categorise the issues associated with the aforementioned questions under four terms, namely, conclusion validity, internal validity, construct validity and external experimental validity. There is widespread agreement that such terms constitute experimental validity (Frankfort-Nachmias & Nachmias, 1992; Judd et al., 1991; Robson, 1993; Wohlin et al., 2000).

5.2 Conclusion Validity

Conclusion validity is concerned with the statistical relationship between the treatments of the independent variable and its effect on the dependent variables, given a certain significance level (Wohlin et al., 2000). There are several issues that may affect the ability to make correct conclusions about such relationships and which are potential threats to making correct conclusions (Cook & Campbell, 1979; Wohlin et al., 2000). However, threats are not always applicable to all experiments. The following sections address some of the threats that are applicable to the quasi-experiments that were undertaken in this study.

5.2.1 Low Statistical Power

In order to ensure that the results derived at the end of the experiment are reliable, the statistical power of the quasi-experiments must be considered in the light of three components (Baroudi & Orlikowski, 1989; Lipsey, 1990; Miller et al., 1997):

i) the significance criterion (α);

ii) the sample size (N); and

iii) the effect size (γ).
The significance criterion (α) is related to the fact that the testing of the stated hypotheses involves the possibility of two different types of errors. On the one hand, it is likely that the test result may suggest the rejection of a null hypothesis when it is actually true; on the other hand, it is possible that the test result may suggest the acceptance of a null hypothesis when it is actually false. These are referred to as Type I and Type II Errors and indicate that the inference made on the basis of the test results is incorrect. Littlefair (2001) in his thesis says:

"...the diagnosis of an alpha [Type I] or beta [Type II] error presupposes that the experimental data was correctly gathered and processed, the incorrect inference having arisen out of the data sampled being by chance a misleading sample."

(p.164).

As proposed by Miller et al. (1997), in this thesis the significance criterion (α) is set at a "prudently low level of 0.05" (p. 287) to protect against Type I error. This means that there is a 1 in 20 chance of incorrectly rejecting a null hypothesis (Miller et al., 1997).

The risk of committing a Type II error is represented by β. The power of a statistical analysis is represented by the probability that the statistical analysis will result in the actual rejection of the null hypothesis when such hypothesis is false and is related to β with the formula 1-β (Land, 2000; Miller et al., 1997). In this thesis, the power of the statistical test is set to 0.8 (allowing for β of 0.2) which, according to Miller et al. (1997), is recommended for research in software engineering.

The size of the sample, commonly referred to by N, is a very important criterion in a quasi-experiment. In this context, Miller et al. (1997) state that:

"As N increases, the probability of error decreases, thus the greater the precision and the higher the chance of rejecting the null hypothesis, assuming that the sample is a representative cross-section of the entire population." (p. 288).

The effect size (γ) refers to the extent to which the phenomenon under study is present in the population: the larger the size of the effect, the greater the possibility that the effect is detectable and, therefore, the null hypothesis is rejected (Miller et al., 1997).
According to Miller et al. (1997) the effect size is typically set to 0.8, which is considered a large effect.

Miller et al. (1997) suggest that in order to achieve the effect size of $\gamma = 0.8$, when the power of the statistical test is set to 0.8 (1-$\beta$) and the significance criterion is set to 0.05 ($\alpha$), data from at least 25 participants are required (Miller et al., 1997). In this study, where $N_{ML1}=133$ and $N_{O1}=39$ and $N_{ML2}=67$, the number of participants is clearly more than 25 (see section 4.2.4.3, table 4.7), which enhances confidence in the reliability of the results. Therefore, the results are not affected by low statistical power.

### 5.2.2 Violated Assumptions of Statistical Tests

As indicated in chapter four, the non-parametric statistical significance test called Mann-Whitney U test was chosen to test the data. This decision is based on the fact that the data (see appendix C, sections 3.3.1, 3.3.2, and 3.3.3) would violate the normality assumption (i.e. were not normally distributed).

This choice is further justified given that the intervention (i.e. $N_{ML1}=133$) and non-intervention (i.e. $N_{O1}=39$) group samples sizes were uneven (see chapter four, section 4.2.4.4). In such circumstances, Douglas & Webster (1999) recommend the choice of non-parametric over parametric tests:

"Non parametric tests were used rather than parametric tests, because they are better suited to unequal group size and non-normality, both of which can seriously bias parametric tests..." (p. 3).

This point is also made by Huck (2000) who also claims that, with nonnormal distributions, non-parametric tests can provide a better protection against Type II errors.

Therefore, it can be concluded that the choice for the Mann-Whitney U test as a non-parametric test is justified, due to the uneven sizes of the samples and non-normal distribution of the data collected from such samples.
5.2.3 Reliability of the Measurements
The conclusion validity of an experiment is also dependent on the reliability of the measurement of the dependent variables. A measurement is said to be reliable if it can be repeated with the same outcome. Measurements can be classified into two broad categories, namely, objective and subjective measurements. Objective measurements (e.g. Lines of Code, page count, etc.) are normally more reliable than subjective measurements (e.g. an opinion or subjective evaluation) because the latter involves more human judgement.

Measurement data in this study was generated by the participants and the reviewers. The participants generated data by filling questionnaires to evaluate the Error Framework or the Catalogue of Errors, etc. In order to minimise any reliability problems, close attention was paid to the wording of the questions to avoid ambiguity.

The reviewers of the artifacts produced by the participants were another source of measurement data. In some circumstances the reviewers had to use their own judgement to determine the error type (see chapter four, section 4.2.5). While most errors do not lend themselves to subjective judgement (e.g. interface errors, object initialisation errors, etc.) there are some (e.g. requirements errors) that can be misinterpreted. For example, it is possible for an inconsistency error to be confused with an ambiguity error. This issue is also raised in Bell & Thayer, (1976). As indicated in chapter four, efforts were made to minimise this potential reliability problem by conducting reviewer meetings before and after the artifacts were reviewed. The objective of such meetings was to ensure that all reviewers had consistent definitions of errors so that the artifacts were reviewed consistently.

5.2.4 Reliability of Treatment Implementation
Treatment implementation concerns how the treatments of the independent variable(s) are applied to the participants. This can be a threat if the treatment is not similarly applied to all participants. There were two treatments of the independent variable, namely, the Catalogue-assisted Java development and Standard Java development. As shown in chapter four, the Catalogue-assisted Java development entailed training the

---

See appendix A, section 1.1 and 1.2, for a complete account of these errors and others.
participants with the Error Framework and the Catalogue of Errors in lecture and workshop sessions. The reliability of treatment implementation threat was minimised by standardising the treatment training materials and using the same person to conduct treatment training lectures. The workshop sessions were conducted by different tutors, who had been instructed to use identical training instruments in the workshop sessions. This ensured that the Catalogue-assisted Java development treatment was similarly applied to all participants.

5.3 Internal Validity
Robson (1993) defines internal validity as that "concerned with the extent to which an experiment establishes that a particular factor or variable has actually caused the effect that is found." (p. 46). Wohlin et al. (2000) add to this definition of internal validity in that it must also ensure that the causal relationship between the independent and dependent variables is not the result of another factor or group of factors over which the investigator has no control or has not measured. Internal validity is subject to a number of threats which might show a causal relationship between independent and dependent variables, when such a relationship does not exist. The potential threats to internal validity are addressed in the following sections.

5.3.1 History Effects
History effects are caused by events that take place during the experiment. Such events may include anything ranging from holiday breaks, where student participants tend to catch up on their work, to an unexpected fire drill during a lecture session, which could disrupt attention during a Catalogue of Errors training session, etc. In order to control for history effects, both the intervention and the non-intervention groups (i.e. JO1 and ML1) were subjected to the treatments during the same time span (Semester 1, 2001) and it is believed that any major events (e.g. holidays) would have affected both groups in the same way. The ML2 group was subjected to the Catalogue-assisted Java development treatment in a different time span (Semester 2, 2001). In addition, all participants used similarly equipped, although different computer laboratories with equal access to identical software packages. It was recognised that complete control over other history effects was impossible.
5.3.2 Maturation and Selection Effects

Maturation effects are related to participant reactions as time passes (Wohlin et al., 2000). It was not possible to rule out the occurrence of maturation effects in this study. For instance, it is possible that the ML1 participants may have become tired or fatigued because the Catalogue of Errors entailed additional work on their part, causing loss of motivation and less effective and efficient performance. Also, it is believed that it is not impossible that the ML1 participants perceived the Catalogue of Errors-related work as something extra, given that it was not part of the unit in previous semesters. This might have caused reluctance, carelessness etc. In either case, the true effect of the Catalogue of Errors may have been underestimated.

On the other hand, it is possible that the ML1 participants may have felt more enthusiastic and motivated because they were learning new things not taught before. It is also possible that the ML1 participants, coincidently, were able to learn faster than others. In either case, they may have performed more effectively and efficiently and the true effect of the Catalogue of Errors may have been overestimated.

It is believed that maturation effects were minimised in this study because the Catalogue of Errors training and other related tasks were included in the unit objectives which students are introduced to once enrolled in a unit. This was expected to ensure that the participants maintained interest throughout the research exercise.

Another threat is related to the non-random assignment of the two treatments which is known as the selection effect. According to Judd, Smith, & Kidder (1991) and Cook & Campbell (1979) selection effects are pervasive in quasi-experiments and “do not permit causal inferences with the same degree of confidence as randomised experiments do”. As a consequence, some internal validity is sacrificed at the expense of practical considerations (see chapter four, section 4.2.4.3). Selection effects were controlled by collecting supplementary evidence to either help establish comparability between the participants or to help explain any possible variation in the results. As a matter of fact, the data collected via the Software Development Background Questionnaire suggests that the ML1, JO1, and ML2 participants are not significantly different. This suggests that the selection effects were minimal.
5.3.3 Testing Threat
The testing threat is concerned with the number of times particular participant responses are measured. If a given participant response is measured more than once, it is likely that the participant performance may be enhanced due to learning effects. It is believed that testing threats have been eliminated in this research because, the participants were not involved in a pre-testing exercise. Participant testing occurred once for each artifact, namely requirements, design and code. It is not believed that this threatened internal validity because the artifacts were different and hence subject to different errors.

5.3.4 Mortality
Mortality is concerned with the number of participants who do not complete the experiment which results in data loss. Experimental results may be affected if the lost data is representative of the total sample. As shown in chapter four, in total, nine participants from ML1 (7) and ML2 (2) did not complete all the software development artifacts. However, given the total size of the samples, mortality would have had only a minimal impact on the internal validity.

5.3.5 Diffusion of Treatment Effects
Diffusion or the imitation of treatment effects occurs when participants from the non-intervention group learn about the treatment and as a result imitate the behaviour of the intervention group. While the lack of diffusion of treatment effects cannot be guaranteed, they were minimised because the participants of the intervention (i.e. ML1) and non-intervention (i.e. JO1) groups were located in two separate campuses.

5.3.6 Compensatory Rivalry and Resentful Demoralization Effects
Typically, compensatory rivalry occurs among the participants of the non-intervention group who may feel as "the natural underdog" (Wohlin et al., p.70) and as a result be motivated to change (i.e. by performing better or worse than expected). As shown in chapter four (see section 4.3.1.1) both intervention (i.e. ML1) and the non-intervention (i.e. JO1) groups were given the same opportunity to acquire the new knowledge that the Catalogue of Errors offered. This helped to minimise any compensatory rivalry or resentful demoralisation effects.
5.4 Construct Validity

Construct validity represents the extent to which the independent and the dependent variables accurately reflect the construct of interest (Judd et al., 1991). In this study, the construct of interest is the impact of knowledge of the Catalogue of Errors on the quality of software development artifacts and developer productivity, and the concrete representation of quality of software development artifacts is measured by the number of errors injected into them by developers during construction (Fenton & Pfleeger, 1997; Kitchenham, 1996; Conte et al., 1986; Diaz & Sligo, 1997; Pfleeger, 1996; Wohlin, 1998). According to Land (2000) and Judd, Smith, & Kidder (1991) the operational definition of the construct must be objective to allow replication by others. Chapter four has clearly defined the dependent and the independent variables and described the instruments to allow replication by others. Yet, construct validity is subject to a number of threats, which are discussed in the following sections.

5.4.1 Hypothesis Guessing

The threat of hypothesis guessing concerns participants who might attempt to work out the purpose and the intended result of the experiment. As a consequence, their behaviour is positively or negatively affected by their guesses about the hypotheses. Attempts were made to minimise this threat on the field experiments by not informing the participants about the dependent and the independent variable information. It is, however, impossible to guarantee that participants were unable to guess what the independent and the dependent variables were.

5.4.2 Hawthorne Effect

The Hawthorne effect is known to affect the performance of participants who are aware that their performance is being observed (Brown, 1954). Kitchenham (1996) explains the Hawthorne effect as when participants perform better in an experiment because of the very fact that they are participating in an experiment. Basili & Weiss (1984) suggest that there is no simple solution for the Hawthorne effect, however, they also say that:

“... if error monitoring is a continuous, long term activity, that is part of the normal scheme of software development, not associated with evaluation of programmer performance, this effect [Hawthorne effect] may become insignificant.” (p.736).
It is believed that the Hawthorne effect was insignificant in the field experiments, because, although the participants were informed that they were participating in experiments, they were not told what data was being collected. Even the collection of error data followed normal university procedures for producing and assessing the participants' artifacts. For example, the errors that were being collected were noted down by their reviewers in terms of comments justifying the deducted marks. Such comments are normally expected in a university environment, when a deliverable (i.e. the artifact(s)) is submitted and flaws that are found are reported (see appendix D). Finally, the project was a long term project taking 12 weeks of a 13 week semester to complete and, as such, possibly exhausting any Hawthorne effect that may have occurred.

5.4.3 Experimenter Expectancies

"Experimenter expectancies" is the term used by Cook & Campbell (1979) and Wohlin et al. (2000) to refer to the potential threat to construct validity from experimenter bias about what they expect from the experiments (Wohlin et al., 2000). In order to minimise this two considerations were made. First, the questions in the questionnaires were designed so that the answers would not be biased to get the answers the experimenter wanted. In addition, participants were able to openly express any opinions in an open-ended question. Second, efforts were made to minimise this threat by involving four different reviewers and conducting meetings before and after the reviews to minimise inconsistencies.

5.5 External Validity

External validity is concerned with the ability of the investigator to generalise "the causal relationship of the construct of the cause, and the effect" (Wohlin et al., 2000, p. 65) outside the scope of the study to the general software engineering population and settings. External validity may be subject to various threats, such as using the wrong participants or using the wrong experimental software development materials. These issues are addressed in the following section.
5.5.1 Participant and Experimental Task Representativeness

The participants who took part in the field experiments were senior students in Computer Science and Software Engineering degrees. The experimental development artifacts that the participants completed are typical industrial software development artifacts (Landis et al., 1990; Landis et al., 1992). Nevertheless, the problems addressed (i.e. the simulators), while being complex and challenging may not be representative of industrial problems. The participants spent twelve weeks of the semester working on them. The fact that the participants were given a deadline to complete software development artifacts is indicative of deadlines within the industry. In addition, participants were not confined in any way and had unlimited access to tools that may be used in industry (e.g. JBuilder, Java Development Kit 1.3.x, Java API Libraries, etc.).

The fact senior students were involved in the study and that they addressed problems of a non-industrial nature may be considered as threats to the external validity of this study. Nevertheless, there are arguments that suggest that generalisation of the findings beyond the immediate scope of the study may be a reasonable proposition.

For example, Wohlin (1998) points out that senior computer science and software engineering students are soon to be first year software developers in an industry where newcomers often work in software development. Therefore, there should be little difference between senior student programmers and first year or junior software developers, and results should be easily generalisable to junior developers in the industry (Wohlin, 1998).

A corollary benefit of any finding is that it may shed light on what the Java development industry can expect when hiring fresh university graduates (Wohlin, 1998; Wohlin et al., 2000).

Further support for the similarity between students and developers is provided in McAndrews (2000), for example:

---

21 The twelve weeks time spent on the artifacts represents the time interval from the time the participants were given the specification to the time they delivered the code. Twelve weeks does not necessarily represent twelve weeks of full time work on the artifacts, because the participants were also students and enrolled in other units and also might have been committed to part-time jobs etc.
"Most software engineers learn how to develop software in college, where the main focus is on learning a language such as C or C++ and writing programs. Students do whatever it takes to write programs, compile them and turn them in. They seldom worry about disciplined methods and quality. Nor are they overly concerned with planning their assignments, with the exception that they probably want to get them done as quickly as possible. The bottom line is that the students do not learn to practice sound development, planning, and quality methods. In addition, the practices they use in college are carried over to their jobs. In many companies, these practices are still considered acceptable." (p. 6).

McAndrews (2000) suggests that the development habits of software engineers are actually formed early when software engineers are educated in college. In addition, there exists empirical evidence suggesting that student participants are similar to professionals in many quantifiable measures, including their approach to developing software (Boehm-Davis & Ross, 1984; Host et al., 2000). Therefore, despite the fact that further refinement does occur in a developer’s approach, basic development habits are formed early (Boehm-Davis & Ross, 1984; McAndrews, 2000).

These arguments, when taken together, suggest that generalisation of the findings of this study to junior or first year Java developers in the industry is reasonable. Nevertheless, the best way to support generalisation is the future replication of this research in different settings.

5.6 Summary and Discussion
The objective of this chapter was to discuss the various validity issues concerning the field or quasi-experiments that were described earlier in chapter four. In the beginning it was argued that it is impossible to know the true relationship between the dependent and the independent variables in the population under study due to the inability of investigators to involve the entire population into the experimental exercise. It was also indicated that in such circumstances, investigators examine the validity of the experiments, which attempts to identify the best approximation of experimental findings about the relationships between dependent and independent variables to the true status of such relationships in the underlying population. In order to identify such
approximation four different aspects of validity were examined. The different aspects of validity that were applicable to the field experiments described in this thesis included, conclusion validity, internal validity, construct validity, and external validity. While several threats to these aspects of validity were identified, arguments about the actions taken to minimise them were also presented.

Wohlin et al. (2000) suggest that different types of research place different priorities on the importance of different aspects of validity. For example, Wohlin et al. (2000) argue that in theory testing research, the most important thing is to show the causal relationship between the dependent and the independent variables (i.e. internal validity). This is followed by an indication that the variables are representative of the construct suggested by the theory (construct validity), which is regarded as the second most important aspect of validity. Wohlin et al. (2000) consider conclusion validity as the third most important, followed by external validity, which in theory testing research “is seldom related to specific settings, population or times to which the results should be generalised” (p. 74). In applied research, however, the priority placed on the importance of each aspect of validity is different. For example, Wohlin et al. (2000) prioritise the different aspects of experimental validity in applied research in decreasing order of importance as follows: internal, external, construct, and conclusion.

The field experiments that were described in chapter four constitute applied research, because they focus on an important area in software engineering, that is the production of quality software artifacts (in terms of fewer errors committed) and developer productivity. The fact that this study is applied research suggests (after Wohlin et al. (2000)) that the different aspects of experimental validity should be prioritised in an order of importance.

In this study internal validity is of greatest importance because the goal of the experiment is to study the relationships between the independent variable (the Catalogue of Errors) and the dependent variables (the number of errors committed, removed, and developer productivity). The generalisation of the experimental findings to a wider context (i.e. the external validity) is the second most important aspect of validity because this study is of limited value if it cannot be applied in a variety of industry and training contexts. The participants and the experimental instruments were
considered as representative of junior developers in an industrial context (Wohlin et al., 2000; McAndrews, 2000). Conclusion validity is the third important aspect of validity because the effect that is expected to accrue has practical importance, and therefore, must be backed by statistical significance.

Finally, construct validity may be seen to be the fourth most important aspect of validity because:

"The applied researcher is relatively less interested in which of the components in a complex treatment that really causes the effect (construct validity)."


The rating of construct validity echoes Basili & Weiss's (1984) statement according to which:

"The results [of an experiment] cannot be used to prove that a particular factor in the development process causes particular kinds of errors, but can be used to suggest that certain approaches, when applied in the environment studied, will improve the development process. The software developer may then be provided with a set of recommended approaches for improving the software development process in his environment." (p. 736).

Therefore, the validity aspects of the field experiments described in this study can prioritised from the most to the least important as follows: internal, external, conclusion, and construct.
6. A Framework to Represent Software Development Errors

6.1 Overview
The objective of this chapter is to address the first research question that was posed in chapter three of this thesis:

*What type of error information is important to help developers learn about errors and how can such information be organised into a generic Error Framework?*

The literature confirms that software errors have been analysed from various perspectives. While some studies have focused on a single perspective, others have attempted to unify multiple perspectives in order to enhance error analysis. The number and type of the perspectives from which an error is viewed determine the range of the developers' knowledge about errors. This, in turn, has a direct impact on the way those errors are committed, prevented, detected and corrected.

This chapter is divided into four sections. In section 6.2, the different standpoints that exist with respect to software development errors are reviewed. In section 6.3, arguments supporting the unification of the various perspectives into a single clear Error Framework are presented. In section 6.4, an Error Framework is proposed, a comparison with other existing frameworks is carried out and the Error Framework is empirically validated. This section also addresses the success factors of research question one (see section 3.4.1, chapter three). Finally, in section 6.5, the chapter is summarised and concluded.

6.2 Software Errors: Different Perspectives
This section addresses two issues. First, what information developers need to know about errors, and, second, why do they need to know such information. An eclectic approach has been taken to address these issues. It is believed that this approach may initially help as a rationale supporting the Error Framework, which is presented later in the chapter. The rationale will also be used to investigate whether the Error Framework can be useful to developers.
6.2.1 How errors are referred to and described?
Various studies use only an error name to refer to software development errors. For example, in her paper, Bytesmiths (1995) lists the names of 54 common Smalltalk errors. This is used as a way to alert developers to possible errors. It is claimed that having such a list could help developers improve the quality of Smalltalk code, because developers would know what not to commit or what to look for when reviewing their code artifacts (Bytesmiths, 1995).

Other works that have provided lists of error names include the work by Beizer (1991), Bashir & Goel (1999) and Binder (2000). In these works comprehensive and well-organised lists of errors are presented. But, unlike Beizer (1991) who presents generic errors, Binder (2000) presents language-specific errors for C++, Java, and Smalltalk. The language-specific error lists are augmented with definitions and descriptions to help understand each individual error (Binder, 2000). Again, Binder’s (2000) motivation for developing such error lists is for them to “... be used as checklists for design and code reviews and in the development of test plans” (p.88). Basili & Perricone (1984) argue that it is very important that an error description be associated with the error name, because the description will not only clarify what the error is all about, but also it “helps for the type of the error to be inferred with a reasonable degree of reliability” (p. 44). In some cases, good error descriptions have even been successfully used to investigate the circumstances leading to errors (Bhandari & Roth, 1993; Jones, 1985).

Mays et al. (1990) propose that in order to aid understanding about errors, an error description should contain an illustrative example of the error. A common method of learning about errors is to include examples (Michalski & Stepp, 1983). For instance, Andersen (1996) extends Overbeck’s work and describes 17 errors exclusive to the C++ language (Overbeck, 1994). In addition to the definitions for each error, examples illustrating the circumstances under which such errors occur are also provided (Andersen, 1996). Andersen (1996) claims that her taxonomy of C++ errors is expected to assist C++ developers test their programs more thoroughly by helping them to create complete sets of test cases.

Yu (1998) takes the issue of the inclusion of examples into descriptions of errors one step further. According to Yu (1998), awareness and knowledge about errors can be
enhanced further, if the illustrative examples consist of two parts, namely, the incorrect, and the correct error example. The incorrect example is an artifact (e.g. section of code) extract containing the error. The correct example is the same artifact extract after the error has been corrected (Yu, 1998). Using correct and incorrect examples to describe an error is considered to enhance learning, because examples and counter examples are used (Ralescu & Baldwin, 1989).

It is widely accepted that using descriptive error names, error definitions, and examples and counter examples helps establish error consciousness among developers. However, if this information is used alone it may be of limited use to developers who are primarily interested in avoiding errors altogether and not just understanding and knowing about them. For example, if developers know about the causes of errors or when during development some errors are likely to be injected, etc. they are more likely to prevent them.

6.2.2 What is the origin of errors?

Some studies have used error origin as a starting point to analyse errors. As the name suggests, error origin pinpoints where in the development of software the error originated and was injected into the artifact. In many cases, origin has been used as a criterion to categorise errors.

Some early studies have denoted the development phase (e.g. requirements specification, low- and high-level design, code, etc.) as the origin where the error is first injected; other studies are more specific and have focused on a specific activity of a development phase where the error is introduced (e.g. initialisation errors are errors that occur during the variable initialisation activity of the code phase); other studies are even more specific, because they clearly map an activity where an error is introduced to the development phase where such activity is performed; another category of studies introduce development paradigm (e.g. object-oriented) specific terminology to refer to the specific activities where errors may be introduced, as the origin of such errors. The following sections overview the studies in the categories outlined above. This is done in order to enhance understanding about error origin and to observe the differences between such studies.
6.2.2.1 Development Phase Origin

Basili & Perricone (1984) report a classification of error origins based on software development phase, which according to them is widely used at the Software Engineering Laboratory (SEL), University of Maryland. This classification includes important development phases (e.g. requirements specification and design) used as origins of errors. This however, is inconsistent in its organisation. Two different criteria have been used, namely the origin of the error (i.e. the phase where the error was introduced), and also the reason(s) leading to the error being introduced (see section 6.2.4)). SEL’s categorisation has been reproduced in table 6.1.

Table 6.1 - SEL’s Error Type Classification (Basili & Perricone, 1984)

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Error Type Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Requirements incorrect or misinterpreted</td>
</tr>
<tr>
<td>B</td>
<td>Functional specification incorrect or misinterpreted</td>
</tr>
<tr>
<td>C</td>
<td>Design error involving several components</td>
</tr>
<tr>
<td>D</td>
<td>Error in design or implementation of single component</td>
</tr>
<tr>
<td>E</td>
<td>Misunderstanding of external environment</td>
</tr>
<tr>
<td>F</td>
<td>Error in the use of programming language/compiler</td>
</tr>
<tr>
<td>G</td>
<td>Clerical error</td>
</tr>
<tr>
<td>H</td>
<td>Error due to previous misdiagnosis of an error</td>
</tr>
</tbody>
</table>

Clearly, SEL’s classification was initially organised using the phase where the error originated as a criterion. However, the last four categories (see figure 6.1, E through to H) encompass errors that occur due to certain reasons or causes (e.g. errors in the use of programming language which mean that the programmer does not have adequate knowledge of a certain aspect of the language).

6.2.2.2 Development Activity Origin

Chillarege et al. (1992) regard development activity as the origin of errors and use it to categorise them (Chillarege et al., 1992). The objective of this classification is to categorise errors on the basis of the semantics of the repair needed to correct them (Fredericks & Basili, 1998) and to capture insights about the errors and their effect on software development using origin (Chillarege, Kao, & Condit, 1991). Chillarege et al. (1992) call their error categorisation orthogonal, which means that no error can possibly fit into more than one category. The categories contained in their orthogonal classification include: function, interface, checking, assignment, timing/serialisation, build/package/merge, documentation, and algorithm errors (Chaar et al., 1993; Chillarege et al., 1992; Chillarege et al., 1991).
Although, the error categories proposed by Chillarege et al. (1992) appear to be self-explanatory a little description on each category may be useful. Function type errors require formal design changes because they affect significant capability, end-users or product or hardware architecture interfaces, and global data structures. Interface errors include errors that result from interacting components, call statements, control blocks, or parameter lists. Checking errors require code changes as they affect proper pre-processing data validation. Assignment errors suggest problems in the initialisation of variables, control blocks, or data structures. Timing and serialisation errors result from the improper management of shared and real-time resources. The build/package/merge category includes errors that affect library systems, change management or version control. Documentation errors correspond to problems that are associated with the written descriptions of user manuals, installation guides, code comments, etc. Finally, algorithm errors include efficiency and correctness issues that may be implemented without the need to request a design change.

Chillarege et al. (1992) have intentionally limited the number of categories to eight, in order to increase the likelihood that a developer will categorise errors identified in a particular project accurately (Fredericks & Basili, 1998). Chillarege et al. (1992) claim that their categorisation allows logical inferences to be made about the development phase where the error was injected without requiring developers to conjecture or speculate about it (Fredericks & Basili, 1998). This is also referred to as process inferencing (Chaar et al., 1993; Chillarege et al., 1992; Chillarege et al., 1991). For example, assignment errors have the tendency to be introduced during the coding phase, whereas algorithm errors tend to be injected during the low-level design phase. Process inferencing, however, relies on the relationship between the development phase and the activities used to represent error origin. Such a relationship is alluded to but not clearly shown in the surveyed publications (Chaar et al., 1993; Chillarege et al., 1992; Chillarege et al., 1991).

A more recent study where development activity is used to represent error origin is conducted by Leszak, Perry & Stoll (2000). They propose their own classification of error origin. According to these authors, the errors in software development may be classified into three large categories, namely, implementation, interface and external
errors. Given that these three categories are very general, they are subdivided into further component categories as shown in table 6.2.

Table 6.2 – Classification of error types (Leszak et al., 2000)

<table>
<thead>
<tr>
<th>Implementation</th>
<th>Interface</th>
<th>External</th>
</tr>
</thead>
<tbody>
<tr>
<td>1: data design/usage</td>
<td>9: data design/usage</td>
<td>16: development environment</td>
</tr>
<tr>
<td>2: resource allocation/usage</td>
<td>10: functionality design/usage</td>
<td>17: test environment (tools/infrastructure)</td>
</tr>
<tr>
<td>3: exception handling</td>
<td>11: communication protocol</td>
<td>18: test environment (test cases/suites)</td>
</tr>
<tr>
<td>4: algorithm</td>
<td>12: process coordination</td>
<td>19: concurrent work (other releases)</td>
</tr>
<tr>
<td>5: functionality</td>
<td>13: unexpected interactions</td>
<td>20: previous (inherited from previous releases)</td>
</tr>
<tr>
<td>6: performance</td>
<td>14: change coordination</td>
<td>21: other</td>
</tr>
<tr>
<td>7: language pitfalls</td>
<td>15: other</td>
<td></td>
</tr>
<tr>
<td>8: other</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Leszak, Perry & Stoll’s (2000) categorisation contains a total of 21 categories, however, these are not all based on the origin of the error (i.e. the activity where the error was injected). In some instances, a category seems to be based on an aspect of the final software product where an error may be detected (e.g. functionality design/usage). In other instances, a category appears to be based on circumstances causing the error to be introduced (e.g. language pitfalls).

6.2.2.3 Development Phase and Activity Origin

Purchase & Winder (1991) provide an error origin categorisation that has received widespread recognition, although it has not been empirically validated (Andersen, 1996; Battig, 1998; Overbeck, 1994). They view error origin in terms of bug history. A bug history indicates the phase in the software development where a bug or error is introduced, whereas the actual bug [error] type represents the activity in the development phase where the error is injected.

Using the notions of bug type and history, nine types of errors are recognised and it is claimed that this categorisation is orthogonal, noncomplex and exhaustive (see table 6.3) (Purchase & Winder, 1991).

---

22 The classification depicted in table 6.2 is augmented with a third factor whose objective is to specify the nature of each error category into one of the possible three values, namely, incorrect, incomplete and other (Leszak et al., 2000).
Grady (1992) reports an error classification scheme developed at Hewlett-Packard. This classification scheme uses both development phase and activity to categorise errors (Grady, 1992). This categorisation scheme has been reproduced in figure 6.1. As the middle layer of figure 6.1 shows, there are different types of errors that may originate at each development phase and activity.  

While the diagram in figure 6.1 shows that each error type may be mapped to a development phase, Fredericks & Basili (1998) criticise Hewlett-Packard's categorisation by suggesting that it may be flawed when used in practice. If historical data about errors are collected, the mapping between errors and phases should allow for development phase evaluation. According to Fredericks & Basili (1998), the mapping between phases and errors in Hewlett-Packard's categorisation is not clear enough and it requires developers to speculate or conjecture about "what occurred at some point in the past" (Fredericks & Basili, 1998, p.14). Admittedly, this could subject the mapping between errors and origin to misclassification problems (Fredericks & Basili, 1998).
Yu (1997; 1998) also advocates error categorisation by development phase and activity where an error may originate. However, only errors in the low-level design and coding phases have been considered. Yu (1997; 1998) introduces an interesting perspective, which is only alluded to by the diagram of Hewlett-Packard's error categorisation scheme (see figure 6.1). According to Yu (1997; 1998) some errors may be introduced via activities performed in the low-level design phase, but they may also be introduced via activities performed in the coding phase. This explains why there is a significant degree of overlap between the low-level design and coding error types as shown in table 6.4.

Table 6.4 – Low-Level Design and Coding Errors (Yu, 1998; Yu et al., 1997)

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Low-Level Design</th>
<th>Coding</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functionality</td>
<td>Yes</td>
<td>Yes</td>
<td>Error in function (e.g. applicability)</td>
</tr>
<tr>
<td>Interface</td>
<td>Yes</td>
<td>Yes</td>
<td>Problem in data control or information passing</td>
</tr>
<tr>
<td>Data</td>
<td>Yes</td>
<td>Yes</td>
<td>Error in data specification (e.g. data type or domain)</td>
</tr>
<tr>
<td>Logic</td>
<td>Yes</td>
<td>Yes</td>
<td>Error in correctness/consistency of computational and control logic</td>
</tr>
<tr>
<td>Input/Output</td>
<td>Yes</td>
<td>No</td>
<td>Error in input/output functions</td>
</tr>
<tr>
<td>Performance</td>
<td>Yes</td>
<td>Yes</td>
<td>Unsatisfactory functional performance (e.g. response time, throughput)</td>
</tr>
<tr>
<td>Maintainability</td>
<td>Yes</td>
<td>Yes</td>
<td>Problem in ease with which maintenance can be performed (e.g. how easy it is to make changes to system)</td>
</tr>
<tr>
<td>Standards</td>
<td>Yes</td>
<td>Yes</td>
<td>Technical/organisational/industry standards not followed</td>
</tr>
<tr>
<td>Human Factors</td>
<td>Yes</td>
<td>Yes</td>
<td>Problem in user satisfaction (e.g. clear/simple interface)</td>
</tr>
<tr>
<td>Syntax</td>
<td>Yes</td>
<td>Yes</td>
<td>Error in structure of expressions in a language</td>
</tr>
<tr>
<td>Testing/Testability</td>
<td>Yes</td>
<td>No</td>
<td>Problem in process of exercising or evaluation a system component</td>
</tr>
<tr>
<td>Documentation</td>
<td>Yes</td>
<td>No</td>
<td>Error found in feature development documentation</td>
</tr>
<tr>
<td>Duplicate</td>
<td>Yes</td>
<td>No</td>
<td>An error/problem that has already appeared</td>
</tr>
<tr>
<td>Not a problem</td>
<td>Yes</td>
<td>No</td>
<td>Error specified in error reports/list that never became a problem in implementation</td>
</tr>
<tr>
<td>Other</td>
<td>Yes</td>
<td>Yes</td>
<td>Other types not specified in the list</td>
</tr>
</tbody>
</table>
Yu's (1997; 1998) categorisation, suggests that it is possible for the same type of error to originate at more than one phase of the software development. For example, errors pertaining to data control or information passing may be introduced during the interface definition activity that occurs in both low-level design and coding phases (see row 2 (interface) of table 6.4).

Hevner's (1997) error categorisation is based on development phase and activity of origin and is quite comprehensive. It contains 21 categories. The proposed categories and a short description for each have been reproduced in table 6.5.

### Table 6.5 – Error Types (Hevner, 1997)

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MR</td>
<td>Marketing requirement: The product or subsystems in the product do not address market needs adequately.</td>
</tr>
<tr>
<td>RF</td>
<td>Requirement functionality: Incorrect and incompatible product features.</td>
</tr>
<tr>
<td>RI</td>
<td>Requirement interface: Incorrect specification of how the product will interact with its environment and/or users.</td>
</tr>
<tr>
<td>RS</td>
<td>Requirements standards: Requirement does not adhere to locally accepted requirements standards.</td>
</tr>
<tr>
<td>DF</td>
<td>Design functionality: Design does not effectively convey what the intended module should do. Functional requirements not satisfied.</td>
</tr>
<tr>
<td>DI</td>
<td>Design interface: Incorrect design of how the product will interact with its environment or users. Interface requirements not satisfied.</td>
</tr>
<tr>
<td>DP</td>
<td>Design process: Inter-process communication: Incorrect interfaces and communications between processes within the product.</td>
</tr>
<tr>
<td>DD</td>
<td>Design data definition: Incorrect design of data structures to be used in the module/product. Data requirements not satisfied.</td>
</tr>
<tr>
<td>DM</td>
<td>Design module: Problems with the control/flow and execution within processes.</td>
</tr>
<tr>
<td>DE</td>
<td>Design error checking: Incorrect error condition checking.</td>
</tr>
<tr>
<td>DS</td>
<td>Design standards: Design does not adhere to locally accepted design standards.</td>
</tr>
<tr>
<td>CI</td>
<td>Code interface/implementation: Problems related to the calling of, parameter definition of, and termination of subprocesses.</td>
</tr>
<tr>
<td>CL</td>
<td>Code logic: Forgotten cases or steps, duplicate logic, extreme conditions neglected, unnecessary function, or misinterpretation errors. Code does not implement design correctly.</td>
</tr>
<tr>
<td>CC</td>
<td>Code computation: Equation insufficient or incorrect, precision loss, or sign convention fault.</td>
</tr>
<tr>
<td>CD</td>
<td>Code data handling: Problems: Initialised data incorrectly, accessed or stored data incorrectly, scaling or units of data incorrect, dimensioned data incorrectly, or scope of data incorrect.</td>
</tr>
<tr>
<td>CS</td>
<td>Code standards: Code does not adhere to locally accepted coding standards.</td>
</tr>
<tr>
<td>DC</td>
<td>Documentation defect.</td>
</tr>
<tr>
<td>HF</td>
<td>Hardware functionality: Product hardware does not provide functionality defined in requirements.</td>
</tr>
<tr>
<td>HI</td>
<td>Hardware interface: Product hardware interfaces do not interface correctly with software or environment. Defect is in hardware implementation.</td>
</tr>
<tr>
<td>TE</td>
<td>Testing environment: Defect is produced by inadequacy in the testing environment, to include testing hardware, testing software, testing procedure, or operator error. The defect is not the product software or hardware.</td>
</tr>
<tr>
<td>OO</td>
<td>Other: This classification should be used sparingly and, when it is used, the defect should be very carefully and extensively described in associated documentation.</td>
</tr>
</tbody>
</table>

The objective of Hevner’s categorisation is to capture essential information about the origin of the error which is expected to aid the “understanding and improvement of software quality” (Hevner, 1997, p.876). The principal advantage of Hevner’s
categorisation is its reliance on historical error data. In a way, such reliant could be regarded as the categorisation's validation. Unlike Yu's categorisations, Hevner encompasses all phases of software development and claims that the proposed categories meet the necessary and sufficient conditions for orthogonality.

6.2.2.4 Paradigm-specific Activity Origin

Tang, Kao, & Chen (1999) propose an activity-based origin error categorisation scheme based on the object-oriented paradigm for software development. This categorisation comprises three broad categories of errors, namely Type I, II, and III errors. The Type I category includes errors that are strongly related to object-oriented features such as inheritance and polymorphism. For example, typical inheritance errors occur during the inheritance definition activity when a subclass object modifies an instance variable of the superclass object, which in turn causes a change in the behaviour of the superclass object (Marick, 1995a, 1995b). Polymorphism errors occur when an object may be bound to different class specifications during run-time (Binder, 2000).

Type II errors include problems that are associated with object management. This category encompasses dangling reference issues which occur when a destroyed object is referred to and with memory usage problems which happen when the memory allocated to an object is not freed up after the object is no longer needed (Tang et al., 1999).

Finally, Type III errors include all other errors that are not associated with objects and which are related to the procedural paradigm (Beizer, 1990). One obvious benefit from Tang, Kao, & Chen's (1999) categorisation is the relative convenience and ease with which errors can be classified. This may be attributed to the fact that only three broad categories are proposed. The drawback is that the adoption of non-informative category names does not enhance the categorisation.

definition and behaviour, ranging from failure to meet class specification to class invariant violation, class size and complexity etc. Message errors include errors such as a message sent to the wrong receiver, parameter mismatches, etc. Instance variable errors encompass missing initialisation errors, out of range errors, incorrect visibility or scoping, etc. The method error category comprises, message mismatch errors, invariant violation errors, incorrect operations, incorrect post-operation states, etc. The inheritance error category consists of inheritance related errors ranging from abstract class instantiation errors to superclass/subclass inconsistency errors, excessively deep or broad class hierarchy errors and incorrect use of polymorphic protocols, circular class graphs, etc. Finally, the integration error category covers errors ranging from deadlocks to incorrect environment interfaces, memory leaks and reuse component errors.

Yilmaz's categorisation attempts to broaden the range of categories of errors. The fact that the category names have been constructed on the basis object-oriented features (e.g. inheritance) or object-oriented program building blocks (e.g. class, method, instance variable) is an advantage of this categorisation. With this categorisation, developers can be made aware of likely pitfalls of the activities of object-oriented program construction. However, there are some flaws in the proposed categorisation. Firstly, the categories are not mutually exclusive, i.e. orthogonal. For example, invariant violation errors can be classified as incorrect post-operation state errors or vice versa. Secondly, the categories rely heavily on the object-oriented paradigm ignoring procedural errors entirely (i.e. Tang, Kao, & Chen's Type II errors). Such errors cannot be ignored because the methods in object-oriented programs are in fact equivalent to procedures and as such subject to procedural errors (Binder, 2000). Finally, any errors that may be introduced in pre-code phases cannot be included anywhere in Yilmaz's (1998) categories. For example, there is no room for requirements ambiguity errors in any of Yilmaz's categories.

6.2.2.5 Paradigm-specific Phase and Activity Origin
Binder (2000) agrees that a categorisation of errors should be based on error origin pinpointed by the development phase and activity where the error is likely to be injected. In his categorisation, Binder (2000) advocates that a development phase that is used to produce an artifact, is typically made up of various activities and it is during such activities that the errors may creep into the artifact. Binder's categorisation is specifically recommended for object-oriented software. In Binder's categorisation an
abstraction level dimension, including, method, class and cluster or subsystem, has been added. The addition of such a dimension makes it easier to isolate error origin. Binder’s categorisation has been summarised in table 6.6. The complete version can be found in Binder, (2000).

<table>
<thead>
<tr>
<th>Phase</th>
<th>Activity</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Abstraction</td>
<td>Method/Class/Cluster</td>
</tr>
<tr>
<td>Design</td>
<td>Refinement</td>
<td>Method/Class/Cluster</td>
</tr>
<tr>
<td></td>
<td>Encapsulation</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Modularity</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Responsibilities</td>
<td></td>
</tr>
<tr>
<td>Code</td>
<td>General</td>
<td>Method/Class/Cluster</td>
</tr>
<tr>
<td></td>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Exceptions</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Instance variable</td>
<td></td>
</tr>
<tr>
<td></td>
<td>definition/use</td>
<td></td>
</tr>
<tr>
<td>Process</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The first category in Binder’s categorisation addresses the requirements phase and encompasses all errors that may be injected when the methods, classes or clusters of classes are specified. The second category comprises errors introduced during the design of the methods, individual classes and clusters of classes. The design phase, however, contains activities, thus it is more sensible to have subcategories of errors that may be introduced during the completion of individual activities of the design phase. A similar sub-categorisation is done for the code phase. The final category is the process category, which is not a development phase per se, but includes errors that are related to the production of documentation across the different phases, throughout software development. Binder claims that his categorisation is eclectic, and is developed on the basis of his own experience and also based on various other widely accepted categorisations, including, Firesmith, (1993); Hayes, (1994); Trausan-Matu, Teplandi, & Barbuceanu, (1991). The principal advantage of Binder’s categorisation is that phases and activities are logical, convenient and reflect a natural way to think of the origin of errors and their use as categorisation criteria. There is no evidence, however, that Binder’s categorisation has been used or validated in practice.

6.2.2.6 Summary and Evaluation of Categories of Error Origin

Having reviewed the above sources, it is clear that there is widespread agreement about the necessity of knowing the origin of errors. In general, the origin is associated with the
development phase and the specific phase activity during which the error originated. Nevertheless, there are also some inconsistencies among the surveyed studies. For example, some studies are comprehensive and recognise all phases of software development and their component activities as a potential origin for software errors (e.g. Hevner, (1997)), others focus only on some of them (e.g. coding in Yu, (1998); Yu et al., (1997)).

Some studies, fail to recognise phase, but they recognise activity. For example, Chillarege et al. (1992) do not recognise the requirements phase as a source of errors per se, but recognise the documentation activity as a lifecycle-encompassing activity, where requirements, design etc. errors might be injected. Some studies focus on error origin categorisations that are paradigm specific (e.g. (Yilmaz, 1998)), while other studies use origin as a dominant criterion for error categorisation (e.g. Chillarege et al., (1992), Binder, (2000), Purchase & Winder, (1991) etc.), and yet others do not (e.g. Leszak et al., (2000) etc.).

In addition to the differences with respect to organisation criteria (i.e. phase versus activity) and development comprehensiveness (i.e. all versus some development phases) the surveyed sources also vary in level of detail. For example, Hevner (1997) and Leszak, Perry, & Stoll (2000) recognise 21 possible origins of errors, whereas Purchase & Winder (1991) and Chillarege et al. (1992) only recognise 8 and 9 possible origins of errors respectively. There exist arguments and counter arguments supporting both choices. On the one hand, fewer categories suggest each individual category representing error origin is described in more general terms. While this makes error classification easier (Chillarege et al., 1992), it may be of limited help to developers in practice, because a single category may contain different types of errors. As a result useful information about error origin may be lost. For example, documentation error by Chillarege et al. (1992) may comprise errors in requirements artifact documentation (in the requirements phase), design artifact documentation (in design phase), code artifact documentation (in code phase) etc. On the other hand, more categories suggest that error origin is described in more specific terms. While this may add to the overhead to error classification, it isolates the exact error origin. For example, definition/use errors occur during the definition and uses of variables in the code phase of software development.
This variability (organization criteria, development comprehensiveness, and level of
detail) may be explained by the specific nature of the projects or the organization-
specific development processes that the reviewed literature has reported. In either case,
the observed variability has not affected the main observation that developers need to
know about error origin (i.e. phase and activity) because this knowledge will help them
become aware of weaknesses in the way they develop artifacts in that phase and
activity. Such awareness will in turn help them prevent errors.

6.2.3 How is the seriousness of errors depicted?
Errors in software affect users in different ways. For example, an application that fails
to implement a function that the user needs (a missing requirement error) cannot be used
unless the missing function is developed. On the other hand, an application which
displays a date in the YYYYMMDD format instead of the required DDMMYYYY will
just cause a slight inconvenience to users without necessarily affecting the application
operation. Thus, it can be said that the second error is less serious than the first error.
Analysing errors from this perspective constitutes error severity analysis. By definition,
error severity describes how serious a failure caused by an error can be. Error severity is
important because it attempts to evidence the impact of the failure caused by the error
during the operation of a software product (Fenton & Pfleeger, 1997; Peng & Wallace,
1993). This awareness helps developers distinguish between high- and low-impact
ersors.

The literature contains various categorisations of errors on the basis of their seriousness
or severity. These categorisations can be classified into two groups. The first group
(Group One) comprises error severity categorisations which contain only 2-3 levels of
severity. The second group (Group Two) encompasses categorisations that comprise 5-6
levels of severity. Representative studies from both groups are reviewed in the
following sections.

6.2.3.1 Group One
According to Doolan (1992) severity can be used to categorise errors in terms of minor
erors, major errors and super-major errors. Minor errors range from spelling mistakes,
through to sentence construction, and clarification issues. Major errors result if software
is produced from the wrong specification or if developers misinterpret vaguely specified requirements or if a requirement is neglected altogether. Finally, super-major errors make the software useless. The presence of super-major errors requires substantial redesign and rewrite of the final software product. Hevner (1997) recognises only two levels of severity, namely major and minor. An error is of major severity if it affects the service of the software. An error is of minor severity if it does not affect the service rendered by the software (Hevner, 1997). However, the error severity categories suggested by Doolan (1992) and Hevner (1997) are too general. One advantage of having broad or general severity levels is that developers have less chance to miscalculate errors when they are analysing them. The drawback, however, is that errors of close but different severity will be included under the same severity level. It follows that categorisations with general severity levels may be less useful in practice.

6.2.3.2 Group Two
Florac (1992) uses the term “criticality” to indicate the degree of disruption that an error may give to users when they encounter it (Florac, 1992). Florac suggests that “criticality” or severity needs to be measured using several levels on a continuum-like scale, however, Florac (1992) defines only the extremes of a continuum of five levels. Level one or the most critical level, includes errors which cause a catastrophic disruption in the use of the software system, and level five or the least critical level being just an annoyance in the use of the software system. The developer is expected to rate criticality or the severity of errors that do not belong to either of the extremes subjectively (Florac, 1992). Florac’s classification provides a more detailed error severity classification. The lack of intervening definitions leaves room for identical errors to be assigned different severity, therefore, the usefulness of Florac’s severity classification may be undermined.

The error severity categorisation provided by Grady (1992) is more comprehensive and detailed. It provides five levels of severity. In addition, Grady’s severity levels are enhanced by recommendations about how to treat errors of each level. Grady recognises critical errors, serious errors, medium errors, low errors, and unclassified errors. A critical error is said to have occurred when the customer is unable to use the software resulting in a critical impact on its operation. Such problems require immediate solution. A serious error exists when the customer is able to use the application, but its
use is severely constrained. In such case, a temporary solution must be provided.

Medium errors force the customer to use the product with limitations as these errors are not critical to its overall operation. With low errors, the customer can avoid the problem and the use of the product with only minor inconvenience. Finally, an unclassified error is an error that has been reported, but is not yet analysed or classified.

Peng & Wallace (1993) concur with the previous authors in the definition of the concept of error severity. They maintain, however, that severity classification depends on the nature of software. It follows that severity classification should be tailored to reflect the domain of software. Despite this, Peng & Wallace propose their own severity classification which is similar to Grady's (1992). They, however, add an additional level of error severity which encompasses errors that occur as a result of the user's misunderstanding of the delivered functionality.

Other error severity classifications have been developed and reported in the literature (Black, 1999; Demirel, 1996; Patton, 2001). They are very similar to those of Grady (1992) and Peng & Wallace (1993).

6.2.3.3 Summary and Evaluation of Categories of Error Severity

Although the error severity classifications reviewed above differ slightly, they do have some elements in common. First, different categorisations use different levels of severity to accommodate the different types of severity. The number of severity levels for the surveyed categorisation systems ranges from two (Hevner, 1997) to six (Peng & Wallace, 1993). This variability may be explained by the fact that they have probably been intended for software systems in different domains where the severity of error varies (Peng & Wallace, 1993), although no indication of this is given in their original sources. Also, the identification of the levels for each categorisation has been established on purely subjective grounds. Apart from Grady's (1992) categorisation, which offers ways to treat errors belonging to a certain severity level, the other categorisations only provide the definitions of each severity level. There do not seem to be any guidelines to minimise possible subjectivity in the categorisation of errors on the basis of severity. Despite the variability, all authors are consistent on the need-to-know requisite of error severity. Severity information is important for developers because it helps them to prioritise their efforts during development when high severity errors
might be involved, to learn more about error characteristics, and therefore, to prevent them from occurring (Leszak et al., 2000; Sullivan & Chillarege, 1991).

6.2.4 What is the cause of errors?
Endres' (1975) seminal work on error causes defines a cause as a certain condition or group of conditions that should have been different for an error not to be introduced into a development artifact (Endres, 1975). This definition of error cause is subscribed to by other researchers as well (e.g. Purchase & Winder (1991), etc). For example, Jones (1985) supports the necessity for the developers to know this information by arguing that:

"... if people [developers] can be reminded of the most common causes for errors made in a task [development activity] just before they start to perform that task, they are less likely to repeat the error." (p.153).

Endres (1975) argues that knowledge of error causes helps future prevention. He proposes six categories of error causes: technological, organisational, historic, group dynamic, individual, and other causes. Technological causes include problem definitions and feasibility and the availability of procedures and tools. Organisational causes comprise issues relating to the availability of information, communication, resources and workload. The third category, historic, includes the impact that previous development history may have on current projects. For example, the way special problems or errors have been handled in the past may affect the way similar problems are handled in the present. Group dynamic causes include issues relating to team members' willingness to cooperate and accept designated roles. The fifth category of error causes, individual, is related to each individual's experience, talent, qualification, etc. Finally, Endres reserves the sixth category for other causes that may not be categorised in the first five or causes that are simply inexplicable. Interestingly, experience with the development of the IBM DOS operating system and the above categorisation of error causes led Endres to conclude that relatively few categories of error causes encompass most errors committed in system programs, confirming a similar conclusion arrived at in Sprohrer & Soloway, (1986). The findings of Endres (1975) and Sprohrer & Soloway (1986) do not seem to be isolated, because they are also confirmed by Pressman (1997) who argues that although many different errors may be
committed by developers, all may be traced to a relatively small number of causes or a predefined set of causes.

Pressman (1997) advocates that proposed cause categories need to be examined in the light of the Pareto Principle, by which "80 percent of the errors can be traced to 20 percent of all possible causes" (Pressman, 1997, p.195). Some of the historical data presented to support Pressman's argument indicate that a few vital causes account for more than half of the introduced errors. These are incomplete or erroneous requirements, misinterpretations of customer communication, and errors in data representation in code artifacts, etc. This is consistent with a similar discussion presented in Davis, (1995) and in Patton, (2001).

A significant contribution is given by Mays et al. (1990) from IBM. Despite being sceptical about categories and categorisations, on the grounds that error categorisation schemes "obscure the details of the error and its cause" (p. 24), Mays et al. propose an error classification scheme which they claim was useful in their Defect Prevention Process at IBM. Five error cause categories were recognised, namely, oversight, education, communication failure, transcription problems, and development process flaws. The first category, oversight, includes errors where for some reason developers overlooked or failed to take into consideration all possible conditions or cases thoroughly. One example might be a situation where a developer fails to realise that a given variable may assume values exceeding a given maximum value. The education category includes errors that are caused by the failure to understand specific aspects of the product or process, presumably due to lack of training in that area. An example of an error in the education category would be a situation whereby a developer did not understand the manner in which a given character variable is initialised by the compiler.

The third category, communication failure, specifies circumstances in which the developer receives incorrect, incomplete or missing information that may be required to produce an error-free artifact. For example, the designer might fail to inform the programmer about a last minute change in the software design. The fourth category, transcription problems, comprises errors caused by the programmers themselves. Examples of transcription errors include typographical errors, omissions due to negligence etc. The fifth category, the process flaw category, includes problems that
may be caused by process flaws. For example, a development process could be flawed if it does not specify that the developers of the software must interact with the clients.

The fifth category, however, is controversial with some scholars trying to promote it and others trying to demote it. This is because sometimes errors in the fifth category may be used as a scapegoat to hide errors that are caused by developers. Mays et al. (1990) suggest that process errors need to be given due weight. That is each individual error cause should be considered on its own merit and then classified. One of the strengths of categorisation by Mays et al. (1990) is that, in addition to the cause-based classification, they propose preventive actions to avoid errors in each specific cause category. These aspects, however, have been discussed in section 6.2.5.

Yu (1998) also believes that error categorisation should be based on causes or as he refers to them root causes. Yu reports the use of a technique called fishbone analysis, otherwise known as Ishikawa diagramming (Kan, 1995), to identify the root causes of coding errors (Yu, 1998). The fishbone analysis incorporates the identification of six major categories of error root causes, namely, execution/oversight, resources/planning, education/training, communication, process/methodology, and product environment. Yu, however, still found that the major root cause categories were too general, and as a consequence, Yu proposed components for each major root cause category that can be acted upon by software engineers. Yu refers to these components as actionable detailed causes. With this, Yu implicitly supports the assertion by Mays et al. that general category names obscure true error cause details (Mays et al., 1990). These categories of causes and their components have been reproduced in figure 6.2 (Yu, 1998).
As figure 6.2 shows, the advantage of Yu’s cause categorisation is that he not only breaks down the major root cause categories into more specific details, but also chooses names for his detailed causes which carry information about the action to be taken to eradicate them in the future. Hence the term actionable detailed root causes.

Error causes have also been categorised by Card (1998) who posits the development of classes of error causes by claiming that they “help identify clusters in which systematic errors are likely to be found” (p.58). Systematic errors include errors that tend to be frequently repeated and account for a large proportion of the errors that are found in a typical software project (Card, 1998). In addition, Card argues that predefined classifications of error causes may help accelerate the causal analysis of errors and suggests four principal categories of error causes.

First, if the methods employed by developers to engineer their applications are incomplete, ambiguous, or wrong, they may cause errors. Even if the right methods are in place, but their use is not widely enforced, they may still be the cause of various errors. Second, errors may also be caused by the use of unreliable and defective tools and software development environments. The third cause of errors is human error due to
lack of adequate training and understanding. The final category comprises incomplete, ambiguous, and defective requirements or inputs. This category is an attempt to introduce error origin information (e.g. incomplete, ambiguous, and defective requirements may occur during the requirements phase). Card (1998) claims that his categorisation not only helps group errors related by cause, but it also helps identify flaws in software development phases.

Another study and classification of error causes is that of Leszak, Perry & Stoll (2000). They also argue that root cause analysis and categorisation is important because it helps derive countermeasures and improvement actions. The novelty of this classification is that it proposes the replacement of one-dimensional root cause classification with a four-dimensional root cause space. According to Leszak, Perry & Stoll existing error cause categorisations are one-dimensional because they only allow for a single unique root cause to be associated with an error. The four-dimensional root cause space classification, on the other hand, allows for an error to be associated with several underlying causes, rather than just one. The four dimensions include human, review, project, and phase root causes (Leszak et al., 2000). Firstly, the human dimension comprises possible human-related causes, such as, change coordination, lack of domain knowledge, lack of system knowledge, lack of tools knowledge, lack of process knowledge, individual mistakes, errors introduced during repair of other errors, communications problem, and inadequate documentation. Secondly, the review dimension encompasses incomplete or missing reviews, inadequate preparation, and inadequate participation. Thirdly, the project dimension includes project specific reasons for error introduction. For example, time pressure, management mistakes, or other product-related influences could constitute project-specific causes. Fourthly, the phase root causes dimension captures the phase or the resulting artifact where the error might have been introduced (e.g. requirements, high-level design, component specification/design, and component coding). Each phase related root cause may be further qualified by selecting one of several attributes highlighting the nature of the cause (e.g. incorrect, incomplete, ambiguous, changed/revised/evolved, not aligned with customer needs, and not applicable). To provide for situations where a cause may be specified which does not belong to any of the four dimensions, Leszak, Perry & Stoll use a distinct Other category where errors may have no underlying cause. Another characteristic is that Leszak, Perry & Stoll recognise multiple causes for errors. This
categorisation system, like the one proposed by Card (1998), integrates error cause with error origin, because the phase dimension indicates the phase in the development where the error might have been injected.

6.2.4.1 Summary and Evaluation of Causes of Errors

The above review of cause-based error categorisations shows considerable similarity and variation. First, all authors acknowledge that developers need to know about error causes. Second, there is agreement that knowledge of error causes may help avoid future recurrence of errors. Third, the authors of the reviewed publications are consistent in the determination of the causes of errors. For example, all studies agree that errors may be attributed to human factors (including communication problems, education, management-related, reviews, etc.), development factors (including tools, methods, procedures, processes, environment, etc.), and project-specific factors (including time, contractor, resources, etc.).

The differences between the classifications relate to the relative importance that is given to a cause in terms of level of detail. For example, Mays et al. (1990) emphasise human-related causes, where they distinguish between oversight, education, communication, transcription causes. Card (1998) and Leszak, Perry, & Stoll (2000), on the other hand, use a single category to encompass all these types of causes. In another instance, while Mays et al. (1990) recognise only the development process category as a potential cause of errors, Card (1998) recognises categories for development methods, development tools and development environment.

In addition, some authors have made attempts to integrate error cause and error origin information into the same classification (e.g. Card, (1998); Leszak et al., (2000)). The classification by Leszak, Perry, & Stoll (2000) is more comprehensive in this regard than that of Card (1998). This is because Leszak, Perry, & Stoll use the generic term [development] phase, to represent any development phase, whereas Card refers specifically to the requirements phase. In both cases, neither Leszak, Perry, & Stoll nor Card focus on the specific activities of a development phase. The effort to use cause (why?) and origin (where?) information collectively to classify errors is a good idea because, if taken together the why (cause) and the where (origin) information may be more accurate in pinpointing errors than this information used alone. However,
categorisations using the joint consideration of error cause and origin have the potential of becoming highly complex for practical use with the extent of complexity depending on the comprehensiveness of the origin and cause classifications.

In any case, the reason for these differences may be attributed to the fact that these cause analyses have been conducted in different organisations\textsuperscript{25}, where different projects are implemented.

6.2.5 How can errors be prevented?

The software engineering community has also focused on what can be done to avert the introduction of errors during software development (Hayes, 1998). This is known as error prevention. Beizer (1990) supports error prevention by arguing that:

"A prevented bug is better than a detected and corrected bug, because if the bug is prevented, there is no code to correct. ... the thinking that must be done to create a useful test can discover and eliminate bugs before they are coded indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception, to specification, to design coding and the rest." (p. 3)

Error prevention constitutes the application of certain measures or guidelines during the different phases of software development. However, there are no single generic cure-all error prevention steps, guidelines or measures. For example, Endres (1975) argues that the measures that one has to undertake in order to prevent errors are "just as varied as the types of errors" (p.148) one can identify.

\textsuperscript{25} e.g. Card, (1998) at Computer Sciences Corporation, Mays et al., (1990) at IBM, and Yu, (1998) at Lucent Technologies, etc.
Mays et al. (1990) argue that while most error preventive measures or guidelines are error specific, the existence of different types of errors\(^6\) and categories of causes of errors suggests that preventive actions may be categorised as well. They organise preventive actions into different types where each type is tightly coupled with an underlying cause category. For example, errors that are caused by oversight require one or more prevention techniques specifically designed to prevent oversight type errors. A summary of the categorisation of preventive actions proposed by Mays et al. (1990) is presented in table 6.7.

Table 6.7 – Types of Preventive Actions by Error Cause Category (Mays et al., 1990)

<table>
<thead>
<tr>
<th>Error Cause</th>
<th>Preventive Actions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oversight</td>
<td>- Checklists and common error lists; - Cross-reference and product logic documentation available online; - Tools to add automatic checks, such as compiler and post-compile module checkers; - Templates or skeletons that guide the creation of a work product; - Permanent reminders and warnings in product documentation; - Reminders in the form of newsletters, memoranda, and reminder notes;</td>
</tr>
<tr>
<td>Education</td>
<td>- Seminars and classes related to the product; - New-hire education checklists; - Tutorial articles in the product-area newsletters; - Education sessions on the new release functions;</td>
</tr>
<tr>
<td>Communications</td>
<td>- Liaison to receive communications from other areas where the product has dependencies and to pass on the information to others in the product area; - Use of a conference disk to pass information to interested parties in a product area; - Enhanced problem-tracking tool to include automatic notification of changes to affected parties;</td>
</tr>
<tr>
<td>Transcription</td>
<td>- Code spelling checker; - Tool that maintains a release's component list and automatically includes it in the design and specification documents and in the build process; - Variable not declared warning in the compiler to check for names that have been misspelled;</td>
</tr>
<tr>
<td>Process flow</td>
<td>- None specified</td>
</tr>
</tbody>
</table>

As table 6.7 shows, Mays et al. (1990) suggest that error causes and prevention measures are closely related to each other. Endres (1975), however, considers error cause and error prevention to be one and the same by stating that:

"We do not distinguish between error cause and error prevention ..., but select the neutral term “error factor” instead. ... [T]he error factors thus specified indicate at the same time what is relevant to the cause of the error, and what could be done in order to prevent the particular error. For example, if we accept

---

\(^6\) Sprohrer & Soloway (1986) distinguish between bug [error] token and bug [error] types. They define an error token to be an instance of an error in a given program. The error type on the other hand comprises all identical error tokens (Sprohrer & Soloway, 1986). A similar distinction is also made in Basili & Perricone, (1984).
as fact that the cause of an error in device handling is to be found in the lack of clarity of the hardware documentation, then this type of error can be avoided by improving the clarity of the hardware documentation." (p.147)

This suggests that there is no need to distinguish between error cause and prevention, because once the cause is examined, the prevention measure is obvious. It should be borne in mind, however, that Endres' categorisation of error types underlies his own classification of error factors.

Unlike Mays et al.'s classification of error causes, which contains only five categories, Endres' categorisation of error factors (cause and prevention) contains 33. Clearly, Endres' analysis is more fine-grained and specific, whereas May. et al.'s causes are more abstract and general. This distinction makes Endres' error prevention measures more obvious once the cause is ascertained. Mays et al.'s categorisation, however, requires the explicit stipulation of prevention measures.

Yu (1998) at Lucent Technologies has introduced a set of technical guidelines with the objective of preventing coding errors. While the prevention guidelines have themselves not been used to categorise errors, they have been specifically tailored to prevent specific code errors. These guidelines have been summarised in a formal document called the Coding Fault Prevention Guidelines (Yu, 1998). Among other things, the Coding Fault Prevention Guidelines provide detailed explanations of the various coding errors along with recommendations on how to prevent them. For example, the guidelines appear in terms of statements such as: "To avoid increment and decrement errors, a set of ( ) have to be used to force the increment to take place on the contents indicated by the pointer" or "To prevent loop boundary condition errors, all loop boundary tests should be carefully examined" etc.

Sometimes a different strategy is used for developers to prevent the underlying errors. Coding fault (error) inspection checklists have been developed and can be formally integrated into the coding phase. Such checklists comprise a set of questions to stimulate thinking during the coding phase. Examples included in Yu, (1998) are:

- "Is this function providing the necessary functionality?"
- "Will the parameter passed provide sufficient data to achieve that functionality?"
• Would it be more efficient to pass a pointer rather than a group of variables or, if those values are to be modified, a data structure?
• Will the return value provide information that the calling function can readily use?” (Yu, 1998, p. 12)

Other strategies to prevent coding errors include Coding Cookbooks, Coding Courses, Peer Checking etc. Yu regards all these error prevention tools as countermeasures for preventing coding errors. Like Mays et al. (1990), Yu advocates the association of root cause to their respective countermeasures. Figure 6.3 has been adopted from Yu (1998) and depicts the association between three major root cause categories to the respective countermeasures. The root causes shown in figure 6.3 are shortlisted from a longer list of causes because they were found to cause 72% of the coding errors in Yu’s study (Yu, 1998).

Leszak, Perry, & Stoll (2000) highlight the importance of error prevention by stating that its resultant effect improves software quality in terms of fewer errors to be found and repaired in later phases of the development process. In order to accomplish error prevention they provide a prioritised set of countermeasures and improvement actions for selected errors caused by human and inadequate review causes (see section 6.2.4). The benefit expected from the countermeasures and improvement actions is to prevent errors or detect them as early as possible in the development process (Leszak et al.,
Some of these countermeasures and improvement actions have been summarised in table 6.8.

<table>
<thead>
<tr>
<th>Countermeasure and Improvement Actions (Leszak et al., 2000)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Countermeasure</strong></td>
</tr>
<tr>
<td>1. Component Specification &amp; Design Documentation</td>
</tr>
<tr>
<td>- extend to ensure contents includes compliance to non-functional and performance requirements</td>
</tr>
<tr>
<td>2. Component Implementation</td>
</tr>
<tr>
<td>- increase usage of static &amp; dynamic code analysis tools</td>
</tr>
<tr>
<td>- improve unit tests</td>
</tr>
<tr>
<td>3. System &amp; Domain Knowledge</td>
</tr>
<tr>
<td>- extend training offers and attendance on architecture and application domain, improve system design skills</td>
</tr>
<tr>
<td>4. Document &amp; Code Reviews</td>
</tr>
<tr>
<td>- analyze review culture and performance, improve review process</td>
</tr>
<tr>
<td>5. Project Management</td>
</tr>
<tr>
<td>- increase process compliance</td>
</tr>
</tbody>
</table>

6.2.5.1 Summary and Evaluation of Prevention of Errors

In summary, the surveyed studies on error prevention are consistent with each other. For example, Mays et al. (1990), Yu (1998), and Leszak, Perry, & Stoll (2000) all agree that activities such as reviews in general or checklist reviews in particular, error prevention guidelines, education and training about processes and development phases and activities, forward and backward traceability between artifacts of software development, and project management activities are crucial to help prevent errors. Mays et al. and Leszak, Perry, & Stoll suggest that errors can also be prevented when automatic tools such as code analysers and unit test tools are used. Mays et al. and Yu also indicate that training with common error lists can be powerful in preventing errors from being injected into software development artifacts. Overall, the surveyed studies are consistent with the message that developers must take preventive actions if they wish to improve the quality of software artifacts. Such action will at best prevent errors altogether from being introduced in software or at worst help detect them early in software development. Moreover, a recurring theme is also the close relationship between the causes of errors and the resultant preventive actions.
6.2.6 How errors manifest themselves?

Another important perspective from which errors can be analysed is the symptom of the error. While an error identifies what is wrong or incorrect with a development artifact, a symptom is the outward manifestation of an error (Fenton & Pfleeger, 1997; Gilb & Graham, 1996). For example, Gilb & Graham (1996) illustrate the concept of error symptom by providing the following example: "...the accounting system produced incorrect subtotals in the monthly account summary report." (p. 345). The "incorrect subtotals" are just the outward manifestation, i.e. the symptom, perhaps of a multiplication rather than addition symbol, which is the error. Florac (1992) uses the term problem as a synonym to the term symptom. He defines a software problem (i.e. symptom) as "a human encounter with software that causes difficulty, doubt, or uncertainty in the use or examination of software" (p. 8). Florac recognises two types of problems namely, dynamic and static. Dynamic problems (i.e. symptoms) occur with an operational system, whereas static problems (i.e. symptoms) occur with the examination of a development artifact (e.g. program listing) (Florac, 1992). The knowledge of error symptoms is important because it helps developers diagnose the underlying errors responsible for those symptoms (Lee & Iyer, 2000).

Chillarege, Kao & Condit (1991) argue that the knowledge of error symptoms and the errors that cause them is important because it helps developers identify different errors with the same symptoms (Binder, 2000; Lee & Iyer, 2000). They propose a set of fifteen symptom descriptors which are used to describe the possible symptoms that software errors may cause. These symptom descriptors have been reproduced in table 6.9.

<table>
<thead>
<tr>
<th>Symptom</th>
<th>Description</th>
<th>Symptom</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>Defect resulted in data being lost</td>
<td>S9</td>
<td>Unavailability of machine affected</td>
</tr>
<tr>
<td>S2</td>
<td>Hang; machine does not respond</td>
<td>S10</td>
<td>Problems uncovered during a build</td>
</tr>
<tr>
<td>S3</td>
<td>Incorrect data was shown on the display</td>
<td>S11</td>
<td>Unverifiable claim (no test case)</td>
</tr>
<tr>
<td>S4</td>
<td>Incorrect message displayed</td>
<td>S12</td>
<td>Functional claim (needs test case)</td>
</tr>
<tr>
<td>S5</td>
<td>Incorrect data output</td>
<td>S13</td>
<td>Performance claim (needs test case)</td>
</tr>
<tr>
<td>S6</td>
<td>Problem during build; library</td>
<td>S14</td>
<td>Architectural claim</td>
</tr>
<tr>
<td>S7</td>
<td>Performance problem (noticeable)</td>
<td>S15</td>
<td>Core dump (memory fault, abort)</td>
</tr>
<tr>
<td>S8</td>
<td>Unpredictable results occur</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

While Chillarege, Kao, & Condit (1991) claim that their symptom list is sufficiently general and detailed to capture all symptoms, the list only contains symptoms that may
be observed when the system is operational, i.e. dynamic problems (Florec, 1992). Static symptoms have been ignored. A similar list of error symptoms has been developed by Hewlett-Packard (HP) and is reported in Fredericks & Basili, (1998); Grady & Caswell, (1987).

Kelsey (1997) also proposes a set of error symptoms. He defines symptoms as software problems that occur when the system is operational. Seven categories of error symptoms are recognised. These have been summarised in table 6.11.

<table>
<thead>
<tr>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>Problem with user-initiated operation</td>
</tr>
<tr>
<td>FN</td>
<td>Problem in a function (discrete code path) supporting an operation</td>
</tr>
<tr>
<td>U1</td>
<td>Problem occurs in the user interface</td>
</tr>
<tr>
<td>IN</td>
<td>Problem occurs in the installation of the product</td>
</tr>
<tr>
<td>UE</td>
<td>Problem is dependent upon the user environment</td>
</tr>
<tr>
<td>SC</td>
<td>The product does not scale to some specific environment</td>
</tr>
<tr>
<td>SCM</td>
<td>Problem occurs due to systems incompatibility</td>
</tr>
</tbody>
</table>

Kelsey's objective is to identify possible symptoms in order to support the testing phase of software development. This is done by pinpointing operational aspects of the software that require more attention from the developers. While it is implied that "more attention" refers to fixing the errors responsible for the symptoms, no direct guidelines are provided on how to associate the responsible error with the resulting symptom(s) (Kelsey, 1997).

The symptom categorisations proposed by Hewlett-Packard, Chilarege, Kao, & Condit (1991) and Kelsey (1997) have a problem. They simply list the symptoms without showing the errors underlying such symptoms. In a survey about difficult-to-correct errors, the lack of such association was one of the most frequently cited reasons (53%) why some errors are difficult to track down (Eisenstadt, 1997):

"The symptom is often far removed in space or time from the root cause, possibly making the cause difficult to detect." (p. 33)

The categorisation of symptoms by Purchase & Winder (1991) (see table 6.10) is an attempt to fix the problems with the previous symptom classifications. It should be
noted that Purchase & Winder (1991) use the term *deviation* to refer to error symptoms. Thus, each error in the proposed taxonomy is associated with a symptom or deviation.

Table 6.10 – Error symptoms (Purchase & Winder, 1991)

<table>
<thead>
<tr>
<th>Error Type</th>
<th>Deviation or Symptom Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Perceptual Error</td>
<td>Resultant program does not solve the problem</td>
</tr>
<tr>
<td>Specification Error</td>
<td>Resultant program cannot meet the specification</td>
</tr>
<tr>
<td>Abstraction Error</td>
<td>Object partitioning is contrived or erroneous and results in unorthogonal, unwieldy protocols. Unnecessary large frameworks are constructed and the inheritance hierarchy may exhibit inconsistency</td>
</tr>
<tr>
<td>Algorithmic Error</td>
<td>Individual method fails in data-dependent way. Failure is related to task concepts, i.e., it may give correct results within certain data domains and fail in others. Inconsistency message protocol for object. The loss of locality due to inheritance can spread the manifestation of these errors</td>
</tr>
<tr>
<td>Reuse Error</td>
<td>Unexpected and undesired behaviour of system-supplied framework, pluggable class, low-level data structures, and/or support mechanisms</td>
</tr>
<tr>
<td>Logical Error</td>
<td>Individual methods fail in one or more data domains</td>
</tr>
<tr>
<td>Semantic Error</td>
<td>Consistent, often catastrophic, method failure; independent of data domain, dependent on coverage. Runtime message search failures. Caught by static binding or strongly typed compilers, dynamic environments often leave it undetected</td>
</tr>
<tr>
<td>Syntactic Error</td>
<td>Resultant program causes compiler execution or consistent, localized deviations from desired behaviour</td>
</tr>
<tr>
<td>Domain Adherence Error</td>
<td>Runtime exceptions, message search failures, spurious errors from remote system classes</td>
</tr>
</tbody>
</table>

Clearly the association between errors and symptoms enhances the usefulness of the categorisation, because developers have the chance to know the actual error that causes a given symptom. Purchase & Winder (1991) argue that the rationale behind this type of classification is that the symptom constitutes an important error diagnostic element that "help[s] to cure existing bugs [errors]" (p. 17). Purchase & Winder's error symptom categorisation has been adopted by Binder (2000).

6.2.6.1 Summary and Evaluation of Categorisation of Error Symptons

In summary, the surveyed studies are similar but they also differ with regard to symptom categorisation. While most of the studies just report symptom lists, Purchase & Winder (1991) go one step further and associate symptoms to their culprit errors. All studies, however, are consistent in concluding that it is necessary for developers to know about error symptoms. This knowledge enables them, in the best case scenario, to recognise the responsible error, or in the worst case scenario, to narrow down the scope of search for the responsible error. In either case, error diagnosis is considerably enhanced.
6.2.7 How can the presence of errors be exposed?

If errors exist in a software development artifact they are dormant until the artifact is activated (i.e. code artifacts are activated via test case execution; pre-code artifacts are activated by using reviews or inspection). During artifact activation, it is likely that the presence of an error is flagged via a symptom. The facilitator that enables software errors to manifest their presence is called an error trigger or simply a trigger. An example of an error trigger may consist of any set of values \( x_1, x_2, \ldots, x_n \) (i.e. the triggers) such that, when passed in as arguments to a function that is meant to return their sum, their product is returned (incorrect result, i.e. the symptom). Clearly, the incorrect computation results from the presence of the error in the implementation of the function as triggered by \( x_1, x_2, \ldots, x_n \). Other examples of triggers, as shown by Chillarege et al. (1992) include, stress or workload-related triggers, timing triggers, exception handling triggers, boundary condition triggers, existing error fixes, etc. Error triggers have been explored in Chaar et al., (1993); Chillarege et al., (1992); Sullivan & Chillarege, (1991).

Different types of errors need different types of triggers to expose their presence (Binder, 2000) suggesting that knowledge about error triggers for each error type is important if developers are to obtain the confidence\footnote{Here, the term "confidence" is being used in the everyday sense of the word and does not relate to the meaning of confidence as described by Hamlet & Taylor, (1990).} that no such error types are lurking in their artifacts. Knowledge about error triggers is also important because it allows developers to get feedback on the effectiveness of error detection approaches, including testing and verification (Chillarege et al., 1992; Fredericks & Basili, 1998; Freimut, 2001). To illustrate this notion, Chillarege et al. (1992) state that:

"Ideally, the defect [error] trigger distribution for field defects should be similar to the defect trigger distribution found during system test. If there is a significant discrepancy between the two distributions, it identifies potential holes in the system test environment." (p.950).

The knowledge about error triggers, is therefore, needed to determine how the presence of errors is exposed (Sullivan & Chillarege, 1991).
The literature suggests that error triggers need to be produced in a systematic way, and therefore, a plethora of approaches (see chapter two) to help developers accomplish this task has been proposed. The approaches that are otherwise known as testing approaches consist of algorithms or heuristics to create error triggers\(^\text{28}\) from the artifact that is being examined (Binder, 2000). The literature also indicates that no one approach can produce error triggers to help expose all possible errors, that is each approach is specialised to produce error triggers whose objective is to expose certain types of errors (Chaar et al., 1993). For example, Beizer (1990; 1995) and Eickelmann & Richardson, (1996) describe many approaches that generate triggers to help identify errors specific to the procedural paradigm. Hayes (1994) and Binder (2000) achieve a similar goal, but with triggers that expose object-oriented specific errors.

The trigger generation approaches described in the above works focus mostly on code artifacts. Error triggers exist for pre-code artifacts too. For example, Chillarege et al. (1992) present a list of triggers that need to be applied to design artifacts. Such triggers include, design conformance, logic/flow, detail understanding, backward and lateral compatibility, consistency and completeness, language dependency, etc. (Chaar et al., 1993). Other studies have focused on the development of ways to identify triggers for pre-code artifacts (Shull et al., 1999; Shull, 1998; Travassos et al., 1999c; Travassos et al., 1999b; Travassos et al., 1999a).

A more comprehensive discussion on the different approaches by which error triggers can be generated has been made in chapter two of this thesis. In this section, however, it is important to emphasise that error triggers are an important perspective from which errors must be analysed. This is because such analysis generates information about errors that the developers need to know (Sullivan & Chillarege, 1991).

\[^{28}\text{Binder's (2000) definition of a test approach includes algorithms and heuristics to create test cases from a representation, an implementation, or a test model. The test case specifies the pretest state of the implementation under test, the test input or conditions (i.e. the error trigger) and the expected result. For simplicity, this discussion assumes triggers to be the sole product of test approaches. This assumption is based on Chaar et al. (1993) who state: "During the unit/function testing stages, a trigger captures the intent behind creating the test case which, when executed, discovered the defect and can therefore potentially be identified by the designer of a test scenario." (p. 1059).}\]
6.2.8 Summary

The above survey has shown that software errors can be analysed from many perspectives. Such perspectives include, name, origin, severity, cause, prevention, symptom, and trigger. The analysis of such perspectives helps complete understanding about errors by generating additional knowledge about them. The previous sections have also highlighted the rationale motivating the need for such knowledge, which is error prevention and early error detection. Clearly, developers would benefit more if errors are analysed from all rather than only few of the above error perspectives. In the following section, the motivation to unify the surveyed error perspectives into an Error Framework is presented. This is followed by a description of the construction of an Error Framework, which is compared to other existing frameworks and empirically validated.

6.3 Motivation for Unifying Error Perspectives into an Error Framework

In order to build high quality software (i.e. software with the fewest possible number of errors), a good understanding of the errors that a developer is likely to introduce during software development is imperative. However, a good understanding about errors can only be accomplished if they are studied in a structured, rather than an ad hoc manner (Sullivan & Chilarege, 1991). It is, therefore, important that before accumulating information about errors, a clear structure or Error Framework is needed to dictate what information about errors needs to be collected and how that information is to be organised.

As discussed in section 6.2, the analysis of errors from different perspectives constitutes what developers need to know about them (i.e. name, origin, severity, cause, prevention, symptom, and trigger.). Therefore, such a framework should accommodate these perspectives. Building such a framework is supported by Kajihara, Amamiya, & Saya (1993) who hold the view that developers need to learn from errors if they are to avoid their future occurrence. Accordingly, the systematic analysis of errors from different perspectives should lead to the establishment of remedial measures whose implementation will prevent errors from recurring (Kajihara et al., 1993; Thelia, 2000).
Florac (1992) advocates analysing errors from multiple perspectives collectively in order to provide a basis for communicating the meaning of the errors descriptively and prescriptively for measurement purposes (Florac, 1992). This was necessitated in 1992 because there was significant variance in the industry in the way that errors were recorded, reported and used to prevent future occurrence. But a similar variability is also reported in the recent survey results in Paulk, Goldenson, & White, (2000), suggesting that the problem has not been addressed yet. In order to standardise the perspectives about errors, Florac (1992) suggests Fenton’s (1991) approach whereby the “who, what, why, when, where, and how” (p. 11) questions are collectively used to obtain information about errors (Fenton, 1991). For example, questions such as: “What was observed? Why did the errors occur?” help identify the symptom and cause perspectives, respectively.

Florac’s (1992) approach is consistent with the approaches presented in Endres, (1975), Jones, (1985) and Mellor, (1992). Endres advocates that complete valid answers to such questions will definitely lead to fewer problems in the future. Jones (1985) answers such questions and motivates them. According to her, a developer should collectively consider at least five main pieces of information about errors. These include error origin, error cause, error avoidance guidelines, error detection guidelines, and error corrective actions. Information about where the error originated is necessary in order to identify the responsible developer or team to which the error can be directed for further analysis. Cause identification is necessary so that the specific problem underlying the errors can be identified. Error avoidance guidelines are needed as a remedy to prevent an error from occurring again in the future. The error corrective actions are needed because they establish specific steps to remove the error. While error avoidance guidelines are normally general in nature and error corrective actions are project-specific. Finally, error detection guidelines are necessary to enable the developer to expose errors that may have not been prevented.

Purchase & Winder (1991) present another argument to substantiate error analysis and description from various perspectives collectively. For example, they maintain that “solely causal [including error cause and error prevention guidelines] taxonomies are academically elegant, but lack pragmatism, while (arguably worse) exclusively diagnostic [including error symptom and error detection guidelines] taxonomies can
help alleviate known bugs [errors], but offer little long-term help.” (p. 17). This shows that errors should be characterised using a combination of the different perspectives.

Freimut (2001) uses the process-inferencing argument as a rationale for the joint consideration of error perspectives. Freimut, however, considers the information generated by analysing an error from any perspective to be an error attribute. It follows that an error can have as many attributes as there are perspectives from which it is analysed. According to Freimut, individual error attributes may be used for software development analysis purposes. Error attributes carry a lot of information which if analysed collectively, may help to characterise the quality of the processes, procedures, etc. that are already in place to produce a software artifact (Freimut, 2001; Hendrickson, 2001). In fact, Kelsey (1997) argues vigorously that the combination of information about error origin, error symptom, and detection provides insights into the capability and effectiveness of the design activities performed. For example, if symptoms (i.e. problems) with user-initiated operations are found during inspection, it may be an indication that unit testing is not sufficiently comprehensive. Similarly, if an algorithmic error is introduced during the design (i.e. the origin of the algorithmic error), it may be a sign that design inspections have been effective in targeting specific types of errors. Other examples are discussed in Kelsey, (1997). Chernak (1996) concurs with Kelsey (1997) but he is more specific and views the necessity to integrate the different error perspectives as tantamount to the direct improvement of verification and validation, rather than the entire software development. In this context he says:

“A defect [error] model [framework integrating different perspectives] provides us the leverage to perform a checklist formal synthesis based on a regression analysis of existing defects.” (p. 867).

A similar argument is also provided in Perry, (1985); and Perry & Stieg, (1993).

In his article, “My hairiest bug war stories”, Eisenstadt (1997) suggests that the collective examination of information from various perspectives helps developers perform debugging activities (i.e. to correctly identify errors more quickly and more thoroughly). For example, Eisenstadt collectively examines three perspectives, namely, origin, root cause and trigger in order to help in the eradication of relatively difficult
errors (Eisenstadt, 1997). A similar argument is also forwarded in Purchase & Winder, (1991).

In summary, integrating the various perspectives into an Error Framework is beneficial because their joint consideration helps developers understand errors better. Such understanding, not only helps developers commit fewer errors, but may also help to organise error information in meaningful ways so that problems in development can be avoided and errors debugged from artifacts.

6.4 Building the Error Framework and Addressing the Success Factors

6.4.1 Overview
The objective of this section is to build an Error Framework in order to address research question one (see section 3.2.1, chapter three). In this section, the success factors of question one (see section 3.4.1, chapter three) are also discussed with reference to the proposed Error Framework. In order to accomplish these objectives, an Error Framework, which is grounded in the discussions of the previous sections of the chapter, is first proposed and its orthogonality is addressed. Comparisons between the proposed Error Framework and other similar frameworks proposed in the literature are made. This is done in order to identify omissions and problems with existing error frameworks and to show that the proposed Error Framework is more comprehensive than the existing frameworks. The usefulness of the various perspectives of the proposed Error Framework is then empirically validated. Empirical validation is crucial because it helps establish whether the different perspectives comprising the Error Framework will be useful in practice. In addition, empirical validation has the potential to help investigators identify parts of the proposed Error Framework that need further improvement or enhancement. The validation was carried out by the ML1 and ML2 participants (see chapter four, section 4.2.2).

6.4.2 Proposing an Orthogonal Error Framework
An Error Framework is proposed comprising the following perspectives: error name, error origin, error severity, error cause, error prevention guidelines, error symptom, and error trigger. These perspectives have been summarised in figure 6.4, as the bottom layer of the hierarchy.
Depending on the information that the error perspectives represent, they can be related to each other. In his seminal work, Endres (1975) argues that there is a close relationship between cause and prevention guidelines, because if steps were to be undertaken to eliminate the condition or conditions that lead to a given error, then such error would then be prevented. Besides, cause and prevention are also related to the origin of errors, i.e. the phase and/or activity of the development where errors occur or are prevented from occurring. The perspectives of error origin, cause, and prevention guidelines are, therefore, collectively categorised under causal perspectives (the middle layer of the hierarchy in figure 6.4).

In addition, error symptoms and triggers are closely related to each other. This is because it takes one or more triggers to expose a symptom. It was, therefore, decided to categorise the perspectives of error symptoms and triggers under diagnostic perspectives (Purchase & Winder, 1991). Finally, the remaining perspectives determine the identity of errors and are thus categorised under identity perspectives. In this context, the Error Framework can be seen as the union of identity, causal, and diagnostic perspectives.

The individual perspectives of the proposed Error Framework (see bottom layer of figure 6.4) constitute an abstraction of the discussions that were carried out earlier in section 6.2 of this chapter. These discussions have shown that the error information that each individual perspective represents is distinct and covers a unique type of information about errors. It follows that, when taken together, the perspectives of the proposed Error Framework are orthogonal. Consequently, the perspectives of the proposed Error Framework satisfy the orthogonality success factor.
6.4.3 Comparison with Existing Work to Assess Comprehensiveness

The objective of this section is to compare the proposed Error Framework with other error frameworks that were found in literature. This comparison is carried out because it helps to identify omissions and problems with the existing error frameworks and to address the comprehensiveness success factor for the proposed Error Framework (i.e. the outcome of research question one). In addition, this comparison helps point out any inconsistencies in existing error frameworks and also that not every existing error framework has been empirically validated. These issues, when taken together, serve as a rationale motivating the need to carry out the comparison with the existing work and to build a comprehensive and empirically validated Error Framework. In order to simplify this comparison, the perspectives of the various error frameworks (including the Error Framework proposed for this study) have been summarized in table 6.11.

Table 6.11 – Summary of Error Frameworks

<table>
<thead>
<tr>
<th>Source</th>
<th>Name</th>
<th>Origin</th>
<th>Severity</th>
<th>Cause</th>
<th>Prevention Guidelines</th>
<th>Symptom</th>
<th>Trigger</th>
<th>Evidence of Validation?</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Endres, 1975)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>(Jones, 1985)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Mays et al., 1990)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Chilarego et al., 1991)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Purchase &amp; Winder, 1991)</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Chilarego et al., 1992)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Mellor, 1992)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Chen, 1996)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>(Kelsey, 1997)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td>✓</td>
</tr>
<tr>
<td>(Yu, 1998)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td>✓</td>
</tr>
<tr>
<td>(Frenz et al., 2001)</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✔</td>
<td>✓</td>
</tr>
<tr>
<td>Proposed Error Framework</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✔</td>
<td>✓</td>
</tr>
</tbody>
</table>

The summary presented in table 6.11 shows clearly that there is a general agreement that errors need to be identified by an error name. Error origin has been stressed by 82% (9 out of 11) of the surveyed works. Error severity has only been marginally used at 27% (3 out of 11). Causes of errors have been given emphasis in 64% of the existing frameworks (7 models out of the surveyed 11). Only 36% (4 out of 11) of the existing error frameworks have been empirically validated.

27 See following section for empirical validation.
frameworks highlight error prevention guidelines as an important error perspective. Error symptoms are used in 45% (5 out of 11) of the surveyed frameworks. 55% of the surveyed sources include error trigger as part of their frameworks. The last column of table 6.11 shows that only 64% of the surveyed frameworks have been empirically validated.

An Error Framework including all the perspectives listed in table 6.11 would obviously be more powerful and comprehensive than one with fewer perspectives (e.g. Endres, 1975; Chillarege et al., 1991; Chillarege et al., 1992; Chernak, 1996; Kelsey, 1997). For example, it would improve on Endres's (1975), Jones's (1985) and Mays et al.'s (1990) framework, which do not include any information about error severity and symptoms.

The perspectives provided by Purchases & Winder (1991) exclude severity, prevention guidelines and error triggers. The perspectives proposed by Mellor (1992) and Freimut (2001) are similar to each other in that they do not account for error prevention guidelines. Yu (1998) while focusing on some causal perspectives (cause and prevention guidelines) ignores diagnostic perspectives altogether (symptom and trigger).

In summary, the discussion about the data of table 6.11 indicates that between 1975 and 2001 errors have not been analysed comprehensively and that the existing error frameworks are quite inconsistent with each other. While most of the authors who have carried out existing work agree on error name and origin, there are inconsistencies with regard to the emphasis that is placed on the remaining perspectives. Clearly, the proposed Error Framework is an attempt to help developers analyse errors comprehensively, and therefore, aims to satisfy the comprehensiveness success factor.

6.4.4 Empirical Validation of Usefulness of Perspectives of Error Framework
The objective of this section is to empirically validate the Error Framework proposed earlier in section 6.4.2. There are three main reasons why this empirical validation exercise should take place. First, empirical validation is an important aspect of software engineering (Tichy, 1998; Tichy et al., 1995). If the proposed Error Framework is to have any practical value, its validation must be carried out.
Second, as was shown in chapter three, the Error Framework that is proposed in this chapter is intended to be used as a template in order to organise information concerning the various error perspectives for different errors in the Catalogue of Errors (see section 3.2.2, chapter three), to be used by participants (i.e. ML1 and ML2 participants). The validation of the Error Framework by the participants will help them create a mental model about how the knowledge in the Catalogue of Errors is organised. Extensive arguments about the benefits of mental models in learning can be found in Borgman, (1999); Stone et al., (1990).

Third, as previously indicated, the various perspectives of the Error Framework were drawn from the literature. However, this does not necessarily mean that the ML1 and ML2 participants regard each perspective of the proposed Error Framework to be useful. If, for example, one or more perspectives were to be consistently rated as not useful, a lot of effort would be wasted if information about such perspectives were to be included in the documentation of errors in the Catalogue of Errors. In addition, the presence of not useful perspectives also signals that the Error Framework needs to be reviewed and rebuild. Therefore, the only way to know whether the participants recognise the usefulness of the different perspectives of the Error Framework is to empirically evaluate the proposed Error Framework before it is actually used as a template to document errors in the Catalogue of Errors. This is why usefulness of the Error Framework perspectives was listed as a success factor for the outcome of research question one (see section 3.4.1, chapter three) and must be addressed.

In this context, the proposed Error Framework can be considered to be useful, if it withstands empirical validation. This was carried out by asking the ML1 (N=133) and ML2 (N=67) participants to fill in the Error Framework Evaluation Questionnaire. This questionnaire requires participants to rate the usefulness of the various perspectives of the proposed Error Framework on a 1 to 5 ordinal scale. In addition, an open-ended question is used to seek suggestions about additional perspectives that the participants thought might be useful. The open-ended question is important because it can help minimise bias in the responses. A sample of the proposed Error Framework Evaluation Questionnaire has been shown in figure 6.5.
The following sections summarise the ordinal scale and open-ended responses provided by the participants in turn.

6.4.4.1 Ordinal Scale Responses

The ordinal scale responses obtained have been summarised in Table 6.12. In order to simplify the discussion, the term error perspective distribution is defined whereby the word perspective represents any of the seven perspectives of the proposed Error Framework. Error perspective distribution will henceforth refer to the distribution of the participant indications about the perceived usefulness of a given error perspective on the 1 to 5 ordinal scale.

Table 6.12 – Frequencies and Descriptive Statistics

<table>
<thead>
<tr>
<th>Error Perspective</th>
<th>1 Useful</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5 Useful</th>
<th>Median</th>
<th>Range</th>
<th>25%</th>
<th>50%</th>
<th>75%</th>
<th>Interquartile Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Name</td>
<td>9</td>
<td>21</td>
<td>32</td>
<td>79</td>
<td>58</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Error Category/Origin</td>
<td>14</td>
<td>14</td>
<td>48</td>
<td>81</td>
<td>45</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Error Severity</td>
<td>31</td>
<td>30</td>
<td>51</td>
<td>58</td>
<td>36</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Error Cause</td>
<td>14</td>
<td>37</td>
<td>65</td>
<td>51</td>
<td>33</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Error Prevention</td>
<td>11</td>
<td>25</td>
<td>53</td>
<td>58</td>
<td>53</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Error Symptom</td>
<td>18</td>
<td>34</td>
<td>57</td>
<td>49</td>
<td>42</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Error Trigger</td>
<td>23</td>
<td>36</td>
<td>59</td>
<td>46</td>
<td>36</td>
<td>3</td>
<td>4</td>
<td>2</td>
<td>4</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>

The values of the median suggest that the participants rated the error name, error origin, and error prevention distributions relatively high at 4 on the 1 to 5 usefulness scale. A

Figure 6.5 – Error Framework Evaluation Questionnaire
A closer look at the interquartile range values suggests that the interquartile range of error origin distribution is 1. On the other hand, the interquartile range for both error name and error prevention distributions is 2. This means that the median of error origin is a better representative of its own distribution than both the medians of error name and error prevention are of theirs (De Vaus, 2002). The remainder of the error perspectives distributions, namely, error severity, error cause, error symptom, and error trigger all have a median of 3 and the same interquartile range of 2.

In summary, the data from table 6.12 is an indication that participants perceive error name, origin, and prevention perspectives to be more useful than the remaining perspectives of the Error Framework. The median of all error perspective distributions is above 3. This fact, along with the fact that the variability of all error perspective distributions is relatively low (i.e. the maximum value of the interquartile range is 2), suggests that all participants recognise the perspectives of the proposed Error Framework as being relatively useful.

### 6.4.4.2 Open-ended Responses

Only 97 participants responded to the open-ended question in the questionnaire (see figure 6.5). Generally, the responses were brief ranging from short sentences to few words. However, they were consistent and, in summary, indicated the need for error description, explanation, or clarification, in terms of examples and/or additional descriptive or illustrative text. This is an indication showing that when errors are documented in the Catalogue of Errors, the error name perspective of the Error Framework must include additional descriptive information (with examples) about the underlying errors. This is consistent with the discussion that was made earlier in section 6.2.1.

### 6.4.5 Building an Error Framework: Summary and Discussion

The objective of section 6.4 was to build an Error Framework. The building of the Error Framework is important because it constitutes the outcome of research question one. The outcome of research question one is deemed to be successful if the Error Framework
Framework satisfies four success factors, namely, orthogonality, comprehensiveness, usefulness, and useability.

The Error Framework was initially derived from the discussions made earlier in the chapter (see section 6.2). It was also argued that the proposed Error Framework constitutes a collection of orthogonal perspectives from which errors can be analysed.

The proposed Error Framework was then compared and contrasted against other similar error frameworks published in literature. This was carried out in order to identify problems with the existing frameworks. Most importantly the existing error frameworks were shown to not be comprehensive and to be inconsistent with each other. Table 6.11 has therefore shown that the proposed Error Framework satisfies the comprehensiveness success factor.

The empirical evaluation of the usefulness of the perspectives of the proposed Error Framework was also examined. This was carried out by the ML1 and ML2 participants (see chapter four) as part of a field study (see chapter three). This is important due to the general need for empirical validation in software engineering and the need to help participants build mental models about the structure of errors documented in the Catalogue of Errors, which is the outcome of research question two. The evaluation of the perspectives of the Error Framework is also important because it helps address the usefulness success factor. The data presented in table 6.12 has shown that, generally, the ML1 and ML2 participants consider the perspectives of the proposed Error Framework to be relatively useful.

The discussion about the useability success factor of the Error Framework perspectives has been deliberately deferred until chapter seven. The evaluation made with reference to the outcome of research question one (i.e. the Error Framework) is a precursor to further work that this research has set out to accomplish. Part of this work constitutes the construction of the Catalogue of Errors (i.e. the outcome of research question two) by using the perspectives of the Error Framework. Consequently, the useability of the Error Framework perspectives can best be addressed during the development of the Catalogue of Errors, which is described in chapter seven.
In summary, while the proposed Error Framework satisfied the needs of this research, it must be said that the Error Framework may need more thorough evaluation and validation, if it were to be used in different contexts or for different purposes. For example, if the purpose of the Error Framework were to be used as a template to collect historical data about errors that have been committed in past software projects, additional evaluation may be required. In such circumstances, it is suggested that further research about the proposed Error Framework may be warranted.

6.5 Chapter Summary

This chapter has presented an Error Framework that can be used as a template to document information about errors from which developers can learn. The Error Framework constitutes the outcome of research question one (see section 3.2.1, chapter 3). Initially, various perspectives about errors were examined. Attention was paid to present each individual perspective as comprehensively as possible. Most importantly, the information resulting by analysing errors from each perspective was argued to be necessary for developers to know if they are to prevent errors from being injected into software. The motivation to unify the identified perspectives into a single Error Framework is another issue that was discussed in the chapter. Finally, the chapter is concluded with the proposal and the validation of an Error Framework with respect to a set of success factors that were outlined earlier in chapter three.
7. Catalogue of Errors Development

7.1 Overview

The goal of this chapter is to describe the development of the Catalogue of Errors. The Catalogue of Errors is important because it constitutes the outcome of question two of this research:

*How can the error framework (developed in question one) be used to catalogue errors that are commonly injected in various software development artifacts?*

The Error Framework developed in chapter six is used as a template to document selected development errors. In order to make the use of the Catalogue of Errors more manageable, the phases of software development were used as criteria to split the Catalogue of Errors into three components, namely, Catalogue of Requirements Errors, Catalogue of Design Errors, and Catalogue of Code Errors (Mays et al., 1990). As the names suggest, each component catalogues errors injected in the respective development phase. This arrangement is illustrated in Figure 7.1.

![Diagram](image)

*Figure 7.1 – Relationship between the Error Framework, the components of the Catalogue of Errors and the Development Phases*

In this chapter, the term Catalogue of Errors is used generically to refer to all components of the Catalogue. However, when issues about specific components are discussed, their designated names are used (e.g. Catalogue of Requirements Errors to refer to the first component of the Catalogue, etc.).
This chapter is divided into six sections. In section 7.2, the field study methodology that was followed to develop the Catalogue of Errors is revisited. Issues relating to the content of the Catalogue of Errors are discussed in section 7.3. In section 7.4, the differences between two trials of the Catalogue of Errors are described (see figure 4.1, chapter four). In section 7.5, the responses provided by the two groups of participants who tried the Catalogue of Errors are compared and contrasted. The responses of the participants are important because they address the success factors of the outcome of research question two (i.e. the Catalogue of Errors). Potential future enhancements of the Catalogue of Errors are discussed in section 7.6. In section 7.7, the chapter is summarised.

7.2 Catalogue of Errors Development Methodology Revisited
The objective of this section is to describe the methodology used to develop the Catalogue of Errors. In chapter three, it was suggested that a field study would be used to build a Catalogue of Errors by reviewing existing literature and using the Catalogue of Errors Evaluation Questionnaires (see section 4.2.5, chapter four) to evaluate the resulting Catalogue empirically and address the respective success factors in the course of its development. Chapter four suggested that two trials of the field study would take place. In this section, the field study methodology is refined further.

The Catalogue of Errors development methodology is based on the Plan-Do-Check-Act (PDCA) quality improvement methodology (also known as the Shewhart/Demming Cycle (Basili & Caldicera, 1995; Shull, 1998)). The typical PDCA methodology is comprised of four steps. The Plan step requires an investigator to identify, justify and design a change in the way something is carried out (e.g. the way software is developed, the way a tool or technique is used, etc.). The Do step entails trialling the change under controlled conditions. The Check step requires the measurement of the results attributed to the change. Finally, in the Act step, the results obtained from the previous step are used to improve the change that was initiated in the Plan step. This sequence of steps is repeated as required. In summary, the PDCA methodology is used to carry out a change, evaluate it, and further improve it.
Figure 7.2 – Applying the PDCA Methodology to develop the Catalogue of Errors

In this thesis, the Catalogue of Errors constitutes a new tool that is expected to be used by participants during the construction of development artifacts. Therefore, the PDCA methodology can be used on the development and improvement of the Catalogue of Errors. Figure 7.2 portrays the way that the PDCA steps were adapted for the development of the Catalogue of Errors. Initially, the Catalogue of Errors was to be developed during the second half of 2000.

Between February and June 2001, the Catalogue of Errors (henceforth referred to as the original version of the Catalogue of Errors) was trialed for the first time in a twelve week software development exercise. During this trial, feedback was collected from the
ML1 participants who used the original version of the Catalogue of Errors. The feedback was used to revise the Catalogue of Errors (the revised version is henceforth referred to as the revised version of the Catalogue of Errors) in order to make modifications and to improve the way the Catalogue of Errors is presented to the participants. The modifications were planned for and incorporated into the revised version of the Catalogue of Errors during June 2001. From July to November 2001, the revised version of the Catalogue was used in a twelve week software development exercise with the ML2 participants. The ML2 participants provided feedback on the revised version of the Catalogue of Errors. The feedback provided by the ML1 and ML2 groups was then compared.

7.3 Content of Catalogue of Errors

While the format of individual errors included in the Catalogue of Errors was dictated by the Error Framework, there were a number of content-related issues that required resolution, such as:

i) What errors would the Catalogue document?

ii) How would the different perspectives of errors, as identified by the Error Framework, be determined? And, what is the evaluation of the useability success factor of the Error Framework?

iii) In what format would the content of the Catalogue be provided to the participants?

Sections 7.3.1 through to 7.3.3 address these issues in turn. Section 7.3.4 discusses the differences between the original and the revised versions of the Catalogue of Errors and section 7.3.5 describes three example errors from the three components of the Catalogue of Errors, namely, the Requirements, Design and Code Errors, respectively.

7.3.1 What errors to document?

There are many errors that developers may commit during the development of requirements, design and code artifacts. Ideally, the Catalogue of Errors should capture all the possible errors that are likely to be injected in development artifacts. However, the time and cost constraints did not permit the documentation of all possible errors. Besides, the nature of the simulator projects attempted by the ML1 and ML2
participants in the field experiments precluded certain errors from occurring. For example, among others, Mitchell (2000) documents an SQL-related error that can occur in the context of a Java program. While such an error would certainly enrich the content of the Catalogue of Code Errors, it was unlikely to be introduced by the participants into their code artifacts because the simulator projects did not have a database dimension (see appendix B, sections 2.1.1 and 2.1.2).

The decision of what errors to document was based on four criteria. Firstly, the selected errors had to be typical. Hence, the selected errors should be expected to occur during the field experiments where the ML1 and ML2 participants would try the Catalogue of Errors. This criterion was imposed in order to ensure that the Catalogue of Errors contained information that would be useful to the participants. Secondly, the range of the selected errors should encompass all phases of development, namely, requirements, design, and code. This criterion is important because the participants were going to go through such phases to develop the respective software artifacts. Thirdly, code errors should be specific to the Java development environment in which the participants would use Java as a programming language. Finally, the selected coding errors should be errors that escape compilation, i.e. not syntax errors. The reason for this criterion is that if errors were identified by the compiler, developers would be alerted to them and would, therefore, have no choice but to learn about them.

Prior experience with similar simulator projects and an informal survey of the literature for errors that satisfy the above criteria allowed the list of errors to be produced. These errors were then documented in the Catalogue of Errors and have been summarised in table 7.1. A list of the surveyed references showing these errors as commonly occurring is presented in the right hand side of the table.
### Table 7.1 — Errors documented in the three components of the Catalogue of Errors

<table>
<thead>
<tr>
<th>Component</th>
<th>Errors</th>
<th>References</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>Greatness; Inconsistencies; Incorrect Facts; Ambiguities</td>
<td>(Davis, 1993a; Lantolle et al., 1998; Smith, 1998) (ESA, 1987; Rombach, 1990; Thayer &amp; Dorfman, 1997)</td>
</tr>
<tr>
<td>Design</td>
<td>Algorithmic Error; Interface Error; Reuse Error; Strong Coupling Error; Weak Cohesion Error</td>
<td>(Edel, Kappel, &amp; Schreif, 1994; McGregor &amp; Sykes, 1992)</td>
</tr>
<tr>
<td>Coding</td>
<td>Abuse of Namespaces Error; Constructor Invoked; Overridden Method Error; Failure to Establish Class Invariant Error; Issues with Honouring Contexts of Methods Inherited by Different Interfaces; Increment and Decrement Operator Error; Object Initialisation Error; Accessing Methods vs Accessing Instance Variables Error; Cast Down the Inheritance Hierarchy Error; Confusing <code>==</code> with equals() Error; Issues with Inner Classes Error; Memory Leaks Error; Thread Deadlock Error</td>
<td>(Dascom, Monk, Keller, &amp; Bohnenberger, 2000; Haggar, 2000; Hunt, 1999; Mitchell, 2000)</td>
</tr>
</tbody>
</table>

#### 7.3.2 Determining the Error Framework Perspectives for Individual Errors

The search of literature on the different perspectives of individual errors indicated that, for some error perspectives information was unsystematic and widely scattered, whereas for other perspectives, the researcher's own judgement had to be used to extrapolate the required information. For example, information about error descriptions, causes, symptoms for most errors in table 7.1 was normally available from the literature. For some errors, however, information about prevention guidelines was not readily available, in which case, such information was derived by studying the causes of the errors or other error-related information. A similar process was also used to generate information about the triggers for errors. Nevertheless, there were two perspectives of the Error Framework, namely, error origin and severity that were found to require special attention.

#### 7.3.2.1 Error Origin

As discussed in chapter six, the origin of errors is associated with the development phase and activity where the error is introduced. While there exists widespread agreement in the literature about the development phases (Binder, 2000; Grady & Caswell, 1987), there are differences about what activities constitute a phase.
(Chillarege et al., 1992; Leszak et al., 2000). The cited references suggest that the activities performed during a development phase in one software project can differ from the activities performed during the same development phase in a different software project. The choice of activities appears to depend on a variety of factors, such as, the application domain of the project, the scope of the software, development paradigm, organisational standards etc. As a consequence, it became necessary to determine the phases and activities that the participants of the development exercise were to undertake. This is important because, once determined, the phases and activities would then have to be mapped against the errors in table 7.1.

On the basis of previous experience with software projects of this nature and the suggestions of Pressman (1997) and Deitel & Deitel (1999), the phases and activities that the participants would go through for the development of their simulator projects were determined. A summary of such phases and activities is provided in table 7.2.

### Table 7.2 – Summary of Development Phases and Activities

<table>
<thead>
<tr>
<th>Requirements Phase</th>
<th>Design Phase</th>
<th>Coding Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition of goals of software</td>
<td>Identification of Classes</td>
<td>Definition of Data types</td>
</tr>
<tr>
<td>Definition of constraints</td>
<td>Identification of attributes</td>
<td>Object Initialisation</td>
</tr>
<tr>
<td>Definition of scenarios</td>
<td>Identification of methods</td>
<td>Data Handling</td>
</tr>
<tr>
<td>Definition of viewpoints</td>
<td>Inheritance definition</td>
<td>Integration</td>
</tr>
<tr>
<td>Stepwise refinement of goals into requirements</td>
<td>Identification of interactions</td>
<td>Documentation</td>
</tr>
<tr>
<td>Narrative of requirements</td>
<td>Definition of algorithms</td>
<td>Testing</td>
</tr>
<tr>
<td>Definition of requirements limitations</td>
<td>Human Computer Interface</td>
<td>Inspection</td>
</tr>
<tr>
<td>Definition of validation criteria</td>
<td>Validation criteria</td>
<td>Inspection</td>
</tr>
<tr>
<td>Inspection</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To ensure that all participants performed all activities of all phases consistently, the activities were mapped and formalised into software development artifact templates that the participants were to follow in order to produce their artifacts at the completion of a development phase. Samples of the requirements, design, and code artifact templates have been included in the appendix B, section 2.1.4. These templates would not only structure the software artifacts of all participants, but would also ensure consistency across participants. The Catalogue of Errors provided in appendix A (see section 1.1).

---

31 Deitel & Deitel “Java How to Program” reference was the main reference used by the participants enrolled in the Internet and Java Programming unit.

32 This would also make the use of page count as a measurement of artifact size (see section 3.4.3.3, chapter three) more meaningful (Bell & Thayer, 1976).
shows the association between the shortlisted errors and the error origin perspective (i.e. phases and activity).

7.3.2.2 Error Severity

When determining the severity of errors, a problem was encountered. The errors that the Catalogue would document were abstractions of errors that were expected to occur in the participants' development artifacts. This made the assignment of severity to errors difficult because certain instances of an error would have a different severity from other instances of the same error. For example, the requirements artifact of the home alarm software system reported in Pressman (1997) requires the specification of a function called configure system. It is assumed that for some reason, the configure system requirement is omitted and not implemented in the final software. This is an omission error whose impact would be serious because the software cannot be used. The requirements artifact of the same home alarm software system also requires a beep sound to be issued every time the user activates or arms the alarm. If the beep sound requirement were not to be implemented, an omission error would be committed, but the severity of this error would be much lower than the severity of the missing configure system function error. The above examples suggest that it is not possible to rate the severity of an error accurately unless the actual instance of the error is known beforehand. Yet, the actual instance of an error that a participant is likely to come across is impossible to determine during the development of the Catalogue of Errors. Consequently, a different mechanism to define severity had to be used.

Thus, the reason why error severity is important was re-considered. The reason for considering the severity of errors in the Catalogue of Errors was to teach participants that some errors have a higher impact than others. The impact affects users of the software where the errors are located. However, developers also are not immune to such errors, because eventually, the developers need to correct them.

Figure 7.3 illustrates the view of Davis (1993a) with regard to different errors injected into different artifacts. This view which is shared by many in the software engineering community (Binder, 2000; Hevner, 1997; etc.) suggests that in general, errors that are injected in earlier artifacts (e.g. requirements artifacts, design artifacts) are more
difficult and costly to correct than errors that are injected in later artifacts (e.g. code artifacts).

![Diagram of error propagation](image)

**Figures 7.3 - Cumulative effects of errors (Davis, 1993a)**

Besides, errors that are injected in earlier artifacts are more likely to have serious consequences to users of software:

"There is growing evidence that requirements errors can lead to serious accidents [on the users]. ... Unfortunately, fixing requirements errors can be extremely costly, especially if errors are detected late in the software lifecycle." (p. 232) (Heitmeyer et al., 1996).

Thus errors injected in earlier artifacts have a higher impact on both users and developers of software and in addition are more costly to fix than errors that are injected in later artifacts. It is accepted that there are exceptions to this rule. Pressman's (1997) home alarm software system beep sound requirement is one example. Also there are examples of errors that are injected in late artifacts which can also have a serious impact on the user and can be very difficult and time consuming to fix (Patrick, 2001). The experience with memory leaks (a code error injected in code artifacts) reported in (Patrick, 2001) illustrates this.

Nevertheless, it must be borne in mind that the intent of the Catalogue of Errors was to impress upon the participants the need to produce error-free artifacts earlier in the
development rather than later. Therefore, it was decided to adapt Grady's (1992) severity classification and assign severity to errors in the Catalogue arbitrarily, while disclosing the grounds for such decision to the participants and notifying them about the exceptions to the rule.

In this context, it was decided to regard requirements errors as critical. Critical errors would include errors that have the potential to make software unusable and must be corrected immediately. It was decided to treat design errors as serious. Serious errors would include errors that have the potential to severely constrain the user of the system. Serious errors require at least a temporary solution. Finally, code errors would be treated as medium errors. Such errors have the potential to limit the use of software although not critically.

7.3.2.3 An Evaluation of the Useability Success Factor of the Error Framework

At this stage the useability success factor (see section 3.4.1, chapter three) of the perspectives of the Error Framework (see chapter six) could be assessed. As indicated earlier in this section most of the perspectives of the Error Framework are usable because they represent information that can be identified and used in building up a Catalogue of Errors. The error origin and error severity perspectives, however, are two exceptions.

As shown in section 7.3.2.1, information about error origin can be identified and easily mapped to errors in the Catalogue. However, it is anticipated that the error origin information of table 7.2 may not be readily used or mapped to all possible errors in any software development environment or organisation. For example, there may be errors that are not associated with any of the activities or phases shown in table 7.2 (Daconta et al., 2000). Besides, in different organisations, different terminology is often used. This is not uncommon, but may complicate the use of the error origin information of table 7.2 and its mapping to the errors in the Catalogue. This suggests that before the error origin information is used to build up a Catalogue of Errors, the development phases and activities (see table 7.2) must be defined and customised to suit the needs of the organisation or the environment where the Catalogue of Errors is expected to be used.
Section 7.3.2.2 has shown that the error severity perspective is a problem as well. It is impossible to assign concise severity to errors in a Catalogue because severity depends on the specific instances of errors, rather than on abstract error information which is what the Catalogue of Errors contains. This has a direct impact on the usability of the error severity perspective.

In summary, the error origin and error severity perspectives may be considered as being relatively less useful as opposed to the remaining perspectives of the Error Framework (see chapter six). This constitutes a limitation of the Error Framework.

7.3.3 Format of the Catalogue of Errors Provided to Participants

In this section, issues relating to the format of the Catalogue of Errors are discussed. There were two ways that the Catalogue of Errors documentation could have been provided to the participants: first, by presenting each error in the Catalogue in a referenced descriptive fashion, i.e. Descriptive Catalogue of Errors (see appendix A, section 1.1), second, by presenting each error in the Catalogue in point or stepwise format, i.e. point form Catalogue of Errors (see appendix A, section 1.2).

Both formats were developed. The objective of the descriptive Catalogue of Errors was to describe the different perspectives for each error as well as to justify such information. The objective of the point-form Catalogue of Errors is to summarise the information provided in the descriptive Catalogue of Errors in order to make such information more practical for use by the participants.

It was decided that the point-form Catalogue of Errors was a better format to provide to the participants. This is because the research exercise with the Catalogue of Errors would take place in the context of a third year unit in a university environment (i.e. field studies and field experiments, see section 4.2.1, chapter four). The participants who took part in the research would be taking the Internet and Java Programming unit whose materials were already provided in point form format. Consequently, there was no reason for the Catalogue of Errors, which was part of the unit, to be presented in a different format. Furthermore, the Catalogue of Errors was to be covered in lectures and workshops, which were designed to clarify any possible misunderstandings and address any issues that the participants might raise.
7.3.4 Original versus Revised Catalogue of Errors: The Differences

As figure 7.2 suggests, two versions of the Catalogue of Errors were developed, namely the original and the revised Catalogue of Errors. The differences between the original and the revised versions of the Catalogue of Errors were dictated by the feedback that was obtained in the free format section of the Catalogue of Errors Evaluation Questionnaires for the Catalogue of Requirements, Design, and Code Errors (see chapter four, section 4.2.5.1). In order to conform with the organisation of this chapter, this section does not discuss participant feedback. This is the subject of section 7.5 of this chapter. Rather, this section examines the aspects in which the responses of the ML1 participants affected the content of the revised Catalogue of Errors. The differences consist of the following:

i) More examples;
ii) Clarification of terms;
iii) More time spent in class.

The first two points outlined above constitute a change in the content of the original version of the Catalogue of Errors. In the revised version, additional illustrative examples were provided and selected examples were explained in a stepwise fashion in lectures and workshops. As for terminology, additional time was spent in lectures and workshops to discuss terms as well as the references for where such terms could be explored further was included in the revised version of the Catalogue of Errors. The third point constitutes a change in the mechanics of the approach and will be covered in section 7.4 of the chapter. The two versions of the Catalogue of Errors can be found in appendix A (see sections 1.1. and 1.2).

7.3.5 Examples

Due to the size limitation of this chapter, only selected sections from the Catalogue of Errors are included. Given that the Catalogue of Errors has three components, namely, Requirements, Design, and Code Errors, it was decided that a single error from each component would be reproduced in this section.
7.3.5.1 Example Requirements Errors: Omission

The presence of omission errors in the requirements artifact indicates that a requirement is missing or that it has not been specified. This results in an incomplete requirements artifact. Schneider, Martin, & Tsai (1992) provide an example of an omission (p. 194):

Requirements Specification Text: In the case of two incoming trains, each of which has access to a siding of adequate length, the slower one is to be routed onto the siding while the faster one continues on the main track.

Omission Error: There is no information about what to do if the two trains are moving at exactly the same speed as measured by the wayside location.

According to Purchase & Winder (1991), if omissions are not prevented or discovered early in system development, the resultant system will not solve the problem it was designed to address (Bell & Thayer, 1976; Purchase & Winder, 1991).

Lamweerde (2000) maintains that goal representation must be integrated into the development of requirements artifacts. This is seen as a way to establish requirements completeness (Lamsweerde, 2001). A goal is an objective that must be achieved by the system under consideration (Lamsweerde, 2000, 2001; Letier & Lamsweerde, 2002). The rationale behind this integration is that the requirements are complete if “they are sufficient to establish the goals they are refining.” (Lamsweerde, 2000, p.7) (Anton & Potts, 1998; Lamsweerde & Letier, 1998). The principal advantage of incorporating goals into the requirements development is that goals provide a criterion for sufficient completeness of a requirements artifact (Lamsweerde, 2001). A second advantage is that goals help identify not only all functional requirements, but also the non-functional ones (Leveson, 1995; Mylopoulos et al., 1999; Nixon, 1993). Another advantage of goal representation is that it greatly facilitates early phases of software

---

33 "A goal corresponds to an objective that the system should achieve through the cooperation of agents [other software component] in the proposed software and in the environment." (p.7) (Lamsweerde, 2000)

34 Functional requirements lead to the particular functions or services that the system is expected to deliver (Lamsweerde, 2001). Functional requirements are stated formally and are enforced during the implementation. (Mylopoulos et al., 1999).

35 Non-functional requirements are global qualities of the proposed software such as flexibility, maintainability, useability, extensibility, performance, security, safety, etc (Lamsweerde, 2001). Non-functional requirements are normally stated informally and are often contentious. For example, performance goals usually interfere with flexibility goals (Mylopoulos et al., 1999).
development, while providing a rationale to the stakeholders of software for the proposed requirements (Lamsweerde, 2001; Lamsweerde & Letier, 1998; Mylopoulos et al., 1999). A fourth advantage is that goal incorporation into requirements artifacts helps avoid irrelevant requirements (Lamsweerde, 2001). Goals can be identified by looking for keywords such as: objective, purpose, intent, concern, in order to, etc. in a user's statements or documentation (Lamsweerde, 2001). Software goal identification is the first step towards avoiding omissions (Mylopoulos et al., 1999).

The second step constitutes the refinement of goals into subgoals. The subgoals are the lower level representation of goals whose accomplishment leads to the achievement of the goals from which the subgoals are refined (Mylopoulos et al., 1999). Anton & Potts (1998) argue that "Goal refinement is intended to reduce the risk of incomplete requirements." (p. 157). Two types of subgoals are recognised. AND-refinement subgoals relate subgoals to a goal in a way that satisfying all subgoals is sufficient for satisfying the goal (Lamsweerde, 2001). OR-refinement subgoals relate subgoals to a goal in a way that satisfying one of the subgoals is sufficient for satisfying the goal (Lamsweerde, 2000). For example, Lamsweerde (2000) defines some requirements of the San Francisco Bay Area Rapid Transit System (BART) (Lamsweerde, 2000, 2001). One of the goals of this system, is to serve more passengers. This goal can be satisfied by accomplishing either one of its two subgoals (OR-refinement), namely, trains must be more closely spaced OR new tracks need to be added. Upon completion of goal refinement, goals and subgoals (depending on the scope of the application, sub-subgoals, etc.) can be organised into a goal traceability tree or matrix which organises goals and the respective subgoals (Lamsweerde, 2001; Pressman, 1997).

Goal conflict identification and resolution constitute the third step in preventing omissions. It is possible that conflicts between goals may exist (Easterbrook, 1994). A goal conflict is a situation whereby the accomplishment of one goal rules out the accomplishment of another (Lamsweerde, 2000). For example, in a scheduling software system, the goal of making timetables publicly available competes with security or privacy goals.

The next step requires the operationalisation of goals into requirements (Mylopoulos, Chung, & Nixon, 1992). Goal operationalisation entails the translation of goals into
operational functions and constraints for the final requirements artifact document. In Loy & Mitchell’s (1990) ATM example the goal to,

Ensure secure Transactions

translates into the following operationalised constrained requirements:

Requirement: Validate User

Required Precondition: ATM Card is used, PIN is used

Required Postcondition: Valid ATM Card is used, Valid PIN is used

Depending on the scope and the nature of the software being developed, the number of goals and requirements may grow considerably. This may exacerbate the organisation of goals and requirements and even encourage the introduction of omission errors. It is, therefore, necessary that goals and requirements be organised into traceability matrices. An example of the traceability matrix between goals and requirements is presented in table 7.3.

Table 7.3 – Goal – Requirements Traceability Matrix

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Ensure Secure Transactions</th>
<th>Keep detailed transaction information</th>
<th>Etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validate User</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate Transaction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Produce receipt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save transactions</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>information into</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>database</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Traceability is discussed in Landis et al., (1992); Palmer, (1997) and the arguments for developing goal-requirements traceability matrices include: 1) that they provide a justification to stakeholders for specified requirements; 2) that they help in the verification and validation of requirements; 3) that they provide an audit trail; and 4) that they help in detecting requirement and goal conflicts.

The symptom of omission errors is the absence of the specification of required elements in the requirements artifact. McGregor (1998a) sheds light on omission error symptoms and the way to expose them by saying:

“A [requirements] model [artifact] is complete (i.e. lacks omission errors) if no required elements are missing. It is judged by determining if the entities in the model describe the aspects of knowledge being modelled in sufficient detail for
the goals of the current portion of the system under development. This judgement is based on the model's ability to represent the required situations and on the knowledge of experts" (p.21).

McGregor's statement implies that, in addition to ensuring that complete requirements have been developed for each goal, omission errors can also be triggered by getting the users of software involved in the review of requirements artifacts. This is because the users of the software are the knowledge experts as they know what the required software should do (Saiedian & Dale, 2000).

Porter, Votta, & Basili (1995) suggest that the presence of omission errors can also be triggered by subjecting the requirements artifact to the following types of questions:

i) Are the described requirements sufficient to meet the system goals?

ii) Have all requirements been provided with the necessary inputs?

iii) Have all undesired systems states and events been considered, and the appropriate responses specified?

Using questions to trigger omissions is also supported by Potts, Takahashi, & Anton (1994) in their Questions, Answers, Reasons technique, and by Lanubile (1998) and Shull (1998).

7.3.5.2 Example Design Error: Weak Cohesion

Cohesion describes the relatedness or connectivity of the different elements (i.e. instance variables and methods within a class or statements within a method) that a designer includes in the same class (Shtern, 2000). If all elements of the class serve the same goal of the problem domain, the class has a strong cohesion, whereas, if the elements of the class are unrelated, the class has a weak (poor or low) cohesion (Shtern, 2000). Strong cohesion is desirable, whereas weak cohesion is not. Eder, Kappel, & Schrefl (1994) recognise three levels of cohesion, namely, method cohesion, class cohesion, and inheritance cohesion (L. C. Briand et al., 1997).
Method Cohesion

Method cohesion shows the relationship of statements included in a method. Method cohesion ranges from coincidental to functional cohesion and is based on Myer's (1978) classical definitions of cohesion (McConnell, 1993; Pressman, 1997; Schach, 1999):

i) **Coincidental cohesion** is the weakest form of cohesion indicating that all the statements that are carried out in the method have nothing in common besides the fact that they reside in the same method.

ii) **Logical cohesion** methods include statements with similar functionality that are included together not because they are related, but because they happen to perform similar functions. For example, a method called `inputAll()` that inputs customer names, employee time-card information, inventory data, etc. would be an example of a method with logical cohesion.

iii) **Temporal cohesion** methods include statements that are combined into a method because they are performed at the same time. For example, a `startup()` method might read a configuration file, initialise a scratch file, set up a memory manager, and show an initial screen. The statements that perform such functionality are placed together simply because they are required at the same time.

iv) **Procedural cohesion** methods include statements that are connected by the same control flow, i.e. carried out in a specified order. For example, if the users of a reporting system like reports to be printed out in a given order, method `printAll()` may include statements that print a revenue report, an expense report, a list of employee phone numbers, and invitations to a party.

v) **Communicational cohesion** methods include statements that make use of the same data but are not otherwise related. For example, method `getNameAndChangePhoneNumber(employee e)` has communicational cohesion if both the name and the phone number are stored in an `Employee` object.

vi) **Sequential cohesion** methods contain statements that are performed in a specific order and share data from step to step. For example, method `getFileDataAndPerformComputations()` would have sequential cohesion, because it would sequentially carry out three different steps, namely, open file, read file, perform computations on the same file.

i) **Functional cohesion** is the strongest form of cohesion where methods possess sequential cohesion and all statements contribute to a single task or objective.
Examples of strong cohesion methods include \texttt{cos()}, \texttt{getCustomerName()}, \texttt{deleteFile()}, \texttt{computeLoanPayment()}, etc.

Robertson (1993) recognises that although functional cohesion is the best form of cohesion in practice, it is not easy to achieve. In fact, it is nearly impossible to develop methods with functional cohesion. In this context, McConnell (1993) advocates that methods with functional, sequential, communicational, and temporal cohesion are acceptable in practice. Methods with procedural, logical and coincidental cohesion are unacceptable (McConnell, 1993).

\textbf{Class Cohesion}

Class cohesion addresses the binding of elements within the same class. Here, the elements of a class are comprised of non-inherited methods and instance variables. Five levels of class cohesion are recognised (Eder et al., 1994):

i) Classes are said to have \textit{separable cohesion} if they contain methods that access none of the class's instance variables, nor invoke any other method within the class. In this case, the class represents multiple unrelated concepts. This is the weakest level of class cohesion. The following example illustrates separable cohesion:

\begin{verbatim}
class Employee {
   String name;
   String address;
   Date birthDate;
   Date hireDate;
   void computeAge() {_,}
   void computeSalary() {_,
   int computeCompanyRevenue(Project p[]) {_,
}
\end{verbatim}

The method \texttt{computeCompanyRevenue()} takes all projects of a company as input parameters and computes the accumulated revenue of that company. It neither accesses any instance variable of \texttt{Employee} nor does it invoke any other methods of \texttt{Employee}. To improve cohesion the method \texttt{computeCompanyRevenue()} should be factored out and included into a different class, e.g. \texttt{company}. 

ii) A multifaceted cohesion class represents multiple related concepts. In such a class at least one method references an instance variable or invokes methods of a different, but related concept of the same class. The following example illustrates a class with multifaceted cohesion:

```java
class Reorder {
    Item reorderedItem;
    Company reorderedForm;
    int discount;
    int quantity;

    public boolean expectedRevenue(){
        Method expectedRevenue() calculates the expected revenue by subtracting the discount given for a company from the price of an item and multiplying this difference by the quantity of reordered items. The discount given depends on the Company. Therefore, the cohesion of the Reorder class can be improved by including discount into class Company:

        class Company {
            String companyName;
            double discount;

            double discount() {} } } }
```

Method expectedRevenue() calculates the expected revenue by subtracting the discount given for a company from the price of an item and multiplying this difference by the quantity of reordered items. The discount given depends on the Company. Therefore, the cohesion of the Reorder class can be improved by including discount into class Company:

```java
class Company {
    String companyName;
    double discount;

    double discount() {} } }
```

iii) A non-delegated cohesion class comprises instance variables that describe only part of the concept that the class is supposed to represent. The following example illustrates a class with non-delegated cohesion:

```java
class Employee {
    String name;
    Date birthDate;
    Project involvedInProject;
    Employee managerOfProject;

    public double computesalary() {} boolean managerIncomeHigherThanAverageInProject() {} } }
```

In class Employee, the instance variables birthDate and involvedInProject depend directly on the instance variable name. However, the instance variable managerOfProject depends directly by the project referred to by involvedInProject and indirectly on the name instance variable. This constitutes non-delegated cohesion. To improve the cohesion of class Employee the instance variable managerOfProject and method
managerIncomeHigherThanAverageInProject() should be delegated to class Project as follows:

```java
class Project {
    Employee managerOfProject;
    Date startDate;
    Date expectedEndDate;
    ...
    boolean managerIncomeHigherThanAverageInProject() {} ...
}
```

iv) **Concealed cohesion** classes include instance variables and methods such that, if regrouped, might form a new distinct and useful class. The following example illustrates a class with concealed cohesion:

```java
class Employee {
    String name;
    String jobProfile;
    int dayOfBirth;
    int monthOfBirth;
    int yearOfBirth;
    int dayOfHire;
    int monthOfHire;
    int yearOfHire;
    ...
}
```

The instance variables describing the various dates may be factored out to a new class `Date` with instance variables `day`, `month` and `year`. The respective instance variables of the class `Employee` are then replaced by two instance variables of type `Date`, namely, `birthday` and `hireDate`, as follows:

```java
class Employee {
    Date birthday, hireDate;
    ...
}
```

v) **Model cohesion** is recognised as the strongest level of class cohesion and "represents a single, semantically meaningful concept without containing methods which should be delegated to other classes or attributes which can be factored out into separate classes" (Eder, Kappel, & Schrefl, 1994, p.29).

In their discussion, Eder, Kappel, and Schrefl (1994) argue that class designs where cohesion is separable, multifaceted, non-delegated and concealed are unacceptable, and should, therefore, be avoided. These classes result in designs that are difficult to understand and reuse and they should be replaced with classes with model cohesion.
Inheritance Cohesion

Inheritance cohesion portrays the binding of all elements in a class. Here class elements are comprised of all methods and instance variables of a class, i.e. inherited and non-inherited. Sometimes inheritance is used solely to avoid data and code duplication between otherwise conceptually unrelated classes. For example, in an Elevator Simulator example class Elevator and Door can be shown to be subclasses of a class called Location. This is the weakest kind of inheritance cohesion and should, therefore, be avoided (also known as ‘code stealing’). The strongest form of inheritance cohesion occurs when subclass-supersclass relationships are dictated by conceptual ('is-a') relationships, i.e. inheritance is used to define specialised child classes. For example, in a university library system both classes students and lecturers are subclasses of class person.

Pressman (1997) suggests that, although in practice it may not be necessary to determine precise cohesion levels, failure to understand and recognise what is acceptable and unacceptable may cause developers to design non-single-minded classes and methods (McConnell, 1993; Robertson, 1993). Classes where cohesion is unacceptable will not only result in designs which are poorly organised, hard to understand, debug, reuse and modify (McConnell, 1993), but as research shows, such designs are also subject to an increased susceptibility to errors (i.e. error proneness) (Basili et al., 1995; V. R. Basili et al., 1998; Briand et al., 1999b). Basili, Briand, & Morasca (1998) suggest that unacceptable types of cohesion (either method, class, or inheritance) can be detected by reviews and/or inspections of design artifacts. If weak cohesion designs are produced, the best remedy is re-designing classes to make them more cohesive (Stern, 2000).

7.3.5.3 Example Code Error: Memory Leaks

In Java, as in other languages, objects occupy memory. Such memory needs to be reclaimed and returned to the operating system when an object has served its purpose and is no longer needed. If memory is not reclaimed and returned to the operating system, the chances are that eventually there will be insufficient memory for the software to run. Consequently, poor performance, software crashes or java.lang.OutOfMemoryErrors are likely to ensue (Henry & Lycklama, 2000;
Patrick, 2001). These situations are commonly known as memory leaks (Flanagan, 2001).

While in other languages (e.g. C++) developers are responsible for writing code that allocates memory to objects and reclaims it, in Java the task of reclaiming memory is performed automatically. The utility program that performs such task is called a garbage collector (Nyland, 1999). A garbage collector runs automatically constantly searching for objects that are unreachable\(^2\) (hence no longer needed) by any other object in the software (Henry & Lycklama, 2000; Nyland, 1999). This has led to the misconception that memory leaks are not possible in Java programs. Unfortunately this is not always the case (Nyland, 1999). The example in figure 7.4 has been adapted from Flanagan, (2001) to illustrate the possibility of memory leaks in a Java program.

```java
class LeakExample {
    public static void main(String args[]) {
        int big_array[] = new int[100000];
        //this queries big_array and obtains a results
        int result = compute(big_array);
        //At this point big_array is no longer needed. However,
        //it will be garbage collected only when the //method
        //returns, because big_array is a local variable.
        //The method, however, will never return, because, it
        //loops infinitely, handling the user's input.
        for (; ;) { handle_input(result); }
    }
}
```

Figure 7.4 – Example of a possible Memory Leak (Flanagan, 2001)

The example in figure 7.4 shows the invocation of method compute(big_array), which queries big_array to obtain a result. After the result is obtained (i.e. after statement int result = compute(big_array); executes), big_array is no longer needed. Yet, big_array will still remain in memory and can only be garbage collected.

\(^2\) In this context, Begic (2001) recognises three types of objects, namely, reachable, resurrectable, and unreachable. Reachable objects are visible to the garbage collector, however, they are still being referred to by other objects in the application. The garbage collector will not attempt to clean such objects (Begic, 2001). Resurrectable objects are not referred to by other objects, but may become reachable when the garbage collector executes the objects’ finalize() method (Begic, 2001). Every object has a finalize() method which is guaranteed "to perform termination house keeping on the object just before the garbage collector reclaims the memory for the object" (Deitel & Deitel, 2002, p. 426). Unreachable objects are objects which cannot be reached and resurrected and they are the primary candidates to be garbage collected (Begic, 2001).
when the method (method main(String args[])) returns. This occurs because big_array is a local variable. Method main(String args[]), however, will never return, because it is supposed to loop infinitely, in order to handle the user's input. This is a situation where an object (big_array) is left unused in memory. Situations like this are likely to cause memory leaks and cannot be resolved by the Java garbage collector (Flanagan, 2001).

Nylund (1999) provides three causes for memory leak errors. Firstly, unwanted object references can cause memory leaks. The easiest way to avoid these memory leaks is to identify objects that are no longer needed and to assign their references to null. For example, in figure 7.4 the statement: big_array = null; could be inserted after the result variable is initialised (i.e. after statement int result = compute(big_array); executes). The act of assigning null to a variable marks the object referenced by the variable for garbage collection (Deitel & Deitel, 2002).

According to Nylund (1999) a second cause for memory leaks is the developers' failure to free native system resources. Native system resources are typically allocated through the Java Native Interface (JNI) by functions that are external to Java implemented in C or C++. For example, Java developers commonly use Abstract Windowing Toolkit (AWT) classes (e.g. Frame, Graphics) and when such classes are no longer needed, they fail to release the system resources reserved for them using the dispose() or finalize() methods.

A third cause of memory leaks is when Java developers reuse third-party libraries (e.g. Java Development Kit (JDK)) or just reuse code developed by other developers. The reused code may already have errors due to the two causes discussed above (Nylund, 1999; Patrick, 2001). One way to avoid existing memory leaks in libraries is to become acquainted with them by checking errors that are published by the developers of such libraries. For example, Sun's Java Developer Connection Bug Database publishes common errors in JDK libraries (Microsystems, 2002).

Patrick (2001) suggests that developers should carefully program Java applications that include structures that are likely to cause memory leaks. These include collection classes, such as hashtab!es, vectors, arrays, etc. (Flanagan, 2001; Patrick, 2001). In such
cases, developers need to study the lifetime of objects of such classes and assign their references to `null` when such objects are no longer needed (Flanagan, 2001; Haggar, 2000). A Java programmer may even explicitly call the garbage collector to execute using method `System.gc()`.

Another structure that commonly introduces memory leaks includes event listeners (Henry & Lycklama, 2000; Patrick, 2001). Event listeners may cause memory leaks when an object is added to an event listener list, but not removed when the object’s usefulness has lapsed (Henry & Lycklama, 2000). Henry & Lycklama (2000) suggest that memory leaks involving lapsed listeners can be avoided by always pairing calls to `addEventtypeListener()` and `removeEventtypeListener()`.

Patrick (2001) reports that objects accessed from static structures are also prone to memory leaks. Static structures include instance variables, methods or even classes, which, once initialised, will stay in memory for as long as the program that defines them stays in memory (Nylund, 1999). This implies that any object that is referred to by a static structure will be kept in memory, although such an object may not actually be needed, thus, increasing the possibility for a memory leak (Nylund, 1999).

In order to avoid memory leaks, Begic (2001) proposes the development of memory profiles of Java software that are suspected to result in memory leaks. A memory profile portrays the memory requirements of the software at various points during its execution (Begic, 2001). According to Begic (2001), the memory profile will help developers localise the memory “hotspots” and, thereby, make informed decisions on further steps to optimise memory consumption. Memory profiles may not only help

---

37 This practice must be adopted with all objects (Flanagan, 2001; Nylund, 1999).
38 However, calls to `System.gc()` do not guarantee the prompt execution of the garbage collector and the order in which any objects marked for garbage collection will be cleared. The call to `System.gc()` is just a suggestion to the Java Virtual Machine for the garbage collector to be called. This suggestion can be ignored (Deitel & Deitel, 2002).
39 Begic (2001) proposes the use of automatic tools such as Rational Purify in order to create memory consumption profiles. Other tools include JProbe, Optimizelt, JInsight, etc. (Henry & Lycklama, 2000; Nylund, 1999). Such tasks, however, can also be performed by either writing specialised classes (e.g. `ObjectTracker` class in Henry & Lycklama, 2000) or by using operating system tools which help observe the memory needs of an application or process or by using Java Development Kit methods. For example, in class `java.lang.Runtime`, methods `freeMemory()` and `totalMemory()` return the amount of unused and total memory, respectively.
prevent memory leak errors but may also expose their presence, if such errors have already been committed (Begic, 2001; Nylund, 1999).

Although Java, with the introduction of the automatic garbage collector has reduced the potential for memory leaks, they have not been eliminated. Misconception about the capabilities of the garbage collector could itself be a cause for memory leaks (Flanagan, 2001; Henry & Lycklama, 2000; Nylund, 1999; Patrick, 2001). Memory leaks can be avoided when awareness of them and the structures that are susceptible to them are established.

7.3.5.4 Examples: Summary and Evaluation

The previous sections have described examples of requirements, design and code errors that were included in the Catalogue of Errors. The remaining errors (see table 7.1) are documented in a similar way in the descriptive Catalogue of Errors (see Appendix A, section 1.1). The examples shown in section 7.3.5.1 through to 7.3.5.3, indicate that, to prevent some requirements and code errors, additional tasks are required to be carried out. For example, in order to prevent an omission, a developer needs to produce a goal-requirements traceability matrix. Similarly, one of the ways to prevent memory leaks requires developers to study the lifetime of objects and assign their references to null, when they are no longer required. Such error prevention tasks are required for all requirements and for some code errors. As a consequence, tasks to prevent these errors from occurring in requirements or code artifacts may require some additional time, over what may be typically required to produce the requirements or code artifacts.

The design error example indicates that there are errors, such as the weak cohesion errors, which do not require additional tasks to be carried out. Awareness about these errors is required if they are to be avoided. The prevention of these errors does not require additional time to be spent by developers during the construction of design artifacts.

In summary, preventing certain errors that are documented in the Catalogue may involve additional time which, in turn, may result in increased cost. In chapter eight it is shown that the fact that some errors require the completion of additional prevention
tasks, has an impact on the productivity of developers when producing software artifacts.

7.3.6 Content of Catalogue of Errors: Summary
In section 7.3, various issues related to the content of the Catalogue of Errors were addressed. Firstly, the decision of what errors to document in the different components of the Catalogue has been explained. Secondly, the way that information about the different perspectives of individual errors was obtained has been described. Here, the usability success factor of the perspectives of the Error Framework was also addressed. Thirdly, decisions about the format of the Catalogue of Errors have been made. Fourthly, the differences between the original and the revised versions of the Catalogue of Errors have been highlighted. Finally, three errors from the three components of the Catalogue have been described and differences highlighted.

7.4 The Two Trials of the Catalogue of Errors
As indicated in chapter four (see section 4.2.5.2) all participants were trained with the Catalogue of Errors in lectures and workshops. Table 7.4 summarises and compares the amount of time spent in training ML1 and ML2 participants with the Catalogue of Errors.

Table 7.4 -- Catalogue of Errors Training Time

<table>
<thead>
<tr>
<th>Training session</th>
<th>ML1 (Time in mins)</th>
<th>ML2 (Time in mins)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Week 1</td>
<td>Lecture 25</td>
<td>Lecture 40</td>
</tr>
<tr>
<td>Week 2</td>
<td>Lecture 25</td>
<td>Lecture 40</td>
</tr>
<tr>
<td>Week 3</td>
<td>Workshop 20</td>
<td>Workshop 30</td>
</tr>
<tr>
<td>Week 4</td>
<td>Lecture 25</td>
<td>Lecture 40</td>
</tr>
<tr>
<td>Week 5</td>
<td>Workshop 20</td>
<td>Workshop 30</td>
</tr>
<tr>
<td>Week 6</td>
<td>Lecture 25</td>
<td>Lecture 40</td>
</tr>
<tr>
<td>Week 7</td>
<td>Workshop 20</td>
<td>Workshop 35</td>
</tr>
<tr>
<td>Week 8</td>
<td>Lecture 25</td>
<td>Lecture 40</td>
</tr>
<tr>
<td>Week 9</td>
<td>Workshop 20</td>
<td>Workshop 35</td>
</tr>
<tr>
<td>Total Time</td>
<td>205</td>
<td>320</td>
</tr>
</tbody>
</table>

During the trial of the original version of the Catalogue of Errors with the ML1 participants, the amount of time spent to train participants was determined by the amount of time a lecturer or tutor would take to cover the materials of the Catalogue of Errors. This was mainly carried out by one-way communication between the lecturer or
tutor and the participants, but sometimes in the workshops, the communication between the tutor and the participants was more interactive.

During the trial of the revised version of the Catalogue of Errors with the ML2 participants, the coverage was more elaborate. Consequently, more time was spent in explaining the details of the documentation of the Catalogue of Errors in lectures. Participants were also invited to express concerns or raise any issues that they had encountered. During the workshops, additional support was provided on using the materials of the Catalogue of Errors.

As table 7.4 indicates, more time was spent on supporting the ML2 participants than the ML1 participants. The additional time and support that was spent in training ML2 participants with the revised version of the Catalogue of Errors was allocated in order to address one of the concerns that the ML1 participants raised after they were trained with the original version of the Catalogue of Errors. This concern suggested that more time was needed on the coverage of the materials of the Catalogue of Errors. A more detailed discussion of this concern and others is addressed in section 7.5.4.

7.5 The Feedback

7.5.1 Overview
The feedback was collected using Catalogue of Errors Evaluation Questionnaires and was separate for the three components of the Catalogue of Errors. Figure 7.5 illustrates the points in the development where feedback was collected.

The feedback for each component of the Catalogue of Errors consists of three separate elements. Firstly, participants were expected to indicate the amount of time they spent to fully assimilate each component of the Catalogue of Errors. This is important because it helps the investigator address the learnability success factor of the outcome of research question two.
Secondly, the participants were expected to express the level of their agreement or disagreement with a set of twelve questions for each of the three components of the Catalogue of Errors. The twelve questions were designed to seek participant feedback with respect to following research question two success factors:

i) Implementation of Error Framework perspectives in the Catalogue of Errors;

ii) Useability of the Catalogue of Errors;

iii) Catalogue of Errors Usefulness to identify and correct errors; and

iv) Overall software development skill improvement.

Finally, an open-ended section is included in all questionnaires requesting the participants to criticise each component of the Catalogue of Errors. The participants were expected to use such sections to express any problems they had encountered and suggest any improvement that could be made to the Catalogue of Errors. Also the information provided by the participants in the open-ended section might help explain the results obtained from the agreement-disagreement questions. The Catalogue of Errors Evaluation Questionnaires have been included in appendix B, section 2.2.1.3.
The subsections that follow examine the three feedback mechanisms for each component of the Catalogue of Errors and its success factors.

7.5.2 Catalogue of Errors Learning Time and Learnability

In addition to the lectures and workshops the participants also spent some time of their own to learn the Catalogue of Errors. They were required to indicate the amount of time that they spent learning the different components of the Catalogue of Errors independently. Table 7.5 summarises the descriptive statistics for the learning time (in minutes) spent by the ML1 and ML2 participants.

Table 7.5 - Descriptive Statistics for the indicated Catalogue of Errors learning time

<table>
<thead>
<tr>
<th>Catalogue of Errors Component</th>
<th>Mean</th>
<th>Standard Deviation</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirements</td>
<td>ML1</td>
<td>ML2</td>
<td>ML1</td>
<td>ML2</td>
</tr>
<tr>
<td></td>
<td>55.7</td>
<td>44.3</td>
<td>20.3</td>
<td>19.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>25</td>
<td>15</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>130</td>
<td>110</td>
</tr>
<tr>
<td>Design</td>
<td>ML1</td>
<td>ML2</td>
<td>ML1</td>
<td>ML2</td>
</tr>
<tr>
<td></td>
<td>75.9</td>
<td>63.7</td>
<td>24.7</td>
<td>17.4</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>155</td>
<td>110</td>
</tr>
<tr>
<td>Code</td>
<td>ML1</td>
<td>ML2</td>
<td>ML1</td>
<td>ML2</td>
</tr>
<tr>
<td></td>
<td>125.7</td>
<td>98.0</td>
<td>40.8</td>
<td>33.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>60</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>195</td>
<td>175</td>
</tr>
</tbody>
</table>

From Table 7.5 it is clear that on average the ML2 participants spent approximately 10-25 minutes less of their own time than the ML1 participants. This difference is consistent with the other descriptive statistics and it is believed to be attributed to the extra support the ML2 participants received during the lecture and workshop sessions (see Table 7.4). It may also be due to the additional illustrative examples that were added to the original version of the Catalogue of Errors (which was used for ML1) to produce a revised version of the Catalogue of Errors (which was used for ML2). These additional examples made the Catalogue of Errors easier to understand and learn, thereby enhancing its learnability. Knowing about the learnability of the different components of the Catalogue of Errors is important because it sheds light on the effort that is required for developers to assimilate this knowledge before it can be used.

7.5.3 Agreement Questions Responses vs Catalogue of Errors Success Factors

As indicated in section 7.5.1, there were twelve questions in three separate questionnaires capturing participants' evaluation of the Catalogue of Requirements,
Design, and Code Errors, respectively. For convenience of presentation, these questions have been collapsed in table 7.6.

The objective of questions Q1 and Q2 is to evaluate the usability of the three components of the Catalogue of Errors. The objective of question Q3 and Q5 is to obtain the participants' responses on whether a particular component of the Catalogue of Errors helped them understand the identity perspectives of the errors (see chapter six, section 6.4.1).

Table 7.6--Questionnaire for Participants' Feedback for the Requirements, Design, and Code Errors Components of the Catalogue

<table>
<thead>
<tr>
<th>Question</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>The Catalogue of Requirements/Design/Code Errors is easy to use.</td>
</tr>
<tr>
<td>Q2</td>
<td>The Catalogue of Requirements/Design/Code Errors is easy to follow.</td>
</tr>
<tr>
<td>Q3</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand requirements/design/code errors.</td>
</tr>
<tr>
<td>Q4</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand the origin of some requirements/design/code errors.</td>
</tr>
<tr>
<td>Q5</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me appreciate the severity of some requirements/design/code errors.</td>
</tr>
<tr>
<td>Q6</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand the cause of some requirements/design/code errors.</td>
</tr>
<tr>
<td>Q7</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand how to prevent some requirements/design/code errors.</td>
</tr>
<tr>
<td>Q8</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand the symptoms of some requirements/design/code errors.</td>
</tr>
<tr>
<td>Q9</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me identify triggers to expose some requirements/design/code errors.</td>
</tr>
<tr>
<td>Q10</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me identify some requirements/design/code errors.</td>
</tr>
<tr>
<td>Q11</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me correct possible requirements/design/code errors.</td>
</tr>
<tr>
<td>Q12</td>
<td>The Catalogue of Requirements/Design/Code Errors has improved my skills to specify higher quality requirements/design/code.</td>
</tr>
</tbody>
</table>

Questions Q4, Q6, and Q7 were aimed at obtaining the participants' feedback on whether the causal perspectives of the Error Framework were implemented properly. Questions Q8 and Q9 focused on the diagnostic perspectives of the Error Framework. Questions Q3 through to Q9 constituted participant evaluation with regard to error identity, causal, and diagnostic perspectives and, therefore, indicated the evaluation of how well the Error Framework perspectives were implemented in the Catalogue of Errors. Questions Q10 and Q11 were aimed at finding out whether the documented error information had been helpful in identifying and correcting errors. Finally, question Q12 was aimed at discovering whether the participants felt that individual components of the Catalogue of Errors had improved their overall software development skills.

250
The clusters of questions shown above were therefore aimed at addressing the success factors of the outcome of research question two. Figure 7.6 summarises the mapping between the agreement questions and the success factors that were being evaluated.

Figure 7.6 – Mapping between Evaluation Questions and Perspectives being Evaluated

The following sections include the distribution of the data provided by the ML1 and ML2 participants in evaluating the six elements of the middle layer of figure 7.6. Median and interquartile range statistics are used because the collected data are ordinal in nature (De Vaus, 2002) (see appendix C, section 3.2.1 for the overall summary of the data).

As indicated in chapter four all questions were evaluated by the participants on a 5-point Likert agreement-disagreement scale, where each point in the scale was represented as follows:

i) strongly disagree: represented by 1,
ii) disagree: represented by 2,
neutral: represented by 3,
agree: represented by 4, and
strongly agree: represented by 5.

Section 7.5.4 examines the responses that were provided to the open-ended question. The compilation of such responses sheds some light into the reasons why the distributions look the way they do.

7.5.3.1 MLI and ML2 Evaluation of Implementation of Error Identity Perspectives

Table 7.7 summarises MLI and ML2 participants' evaluation of the way the error identity perspectives were implemented in the Catalogue of Requirements, Design, and Code errors.

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Median</td>
<td>Interquart. Range</td>
<td>Median</td>
</tr>
<tr>
<td>Q3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLI</td>
<td>4.0</td>
<td>2.0</td>
<td>4.0</td>
</tr>
<tr>
<td>ML2</td>
<td>4.0</td>
<td>0.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Q5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MLI</td>
<td>4.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>ML2</td>
<td>8.0</td>
<td>1.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Key:

Q3: The Catalogue of Requirements/Design/Code Errors has helped me understand requirements/design/code errors.
Q5: The Catalogue of Requirements/Design/Code Errors has helped me appreciate the severity of some requirements/design/code errors.

The median values (4.0) indicate that both MLI and ML2 participants agree that the Catalogue of Errors helped them understand the information about the documented errors (Q3). It was surprising to see that the ML2 participants were neutral in their indications that the Catalogue of Requirements Errors helped them understand about the severity of requirements errors (Q5) (median 3.0), while agreeing that the other components of the Catalogue of Errors helped them understand about the severity of non-requirements errors (medians 4.0). The MLI participants agreed with Q5 for the Catalogue of Requirements Errors (median 4.0), but were only neutral with Q5 for the Catalogue of Design and Code Errors (medians 3.0). In both cases (Q3 and Q5), the
medians of the ML2 distributions are better representatives than the medians of the ML1 distributions due to lower interquartile ranges (see table 7.7).

7.5.3.2 ML1 and ML2 Evaluation of Implementation of Error Causal Perspectives
Table 7.8 summarises ML1 and ML2 participants' evaluation about the way the error causal perspectives are implemented in the Catalogue of Requirements, Design and Code Errors.

Table 7.8 – ML1 and ML2 Participants' Evaluation of Error Causal Perspectives in the Catalogue of Errors

| Q4: The Catalogue of Requirements/Design/Code Errors has helped me understand the origin of some requirements/design/code errors. | Q6: The Catalogue of Requirements/Design/Code Errors has helped me understand the cause of some requirements/design/code errors. | Q7: The Catalogue of Requirements/Design/Code Errors has helped me understand how to prevent some requirements/design/code errors. |

<table>
<thead>
<tr>
<th>Table 7.8 – ML1 and ML2 Participants' Evaluation of Error Causal Perspectives in the Catalogue of Errors</th>
<th>Catalogue of Requirements Errors</th>
<th>Catalogue of Design Errors</th>
<th>Catalogue of Code Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>Median</td>
<td>Interquart. Range</td>
<td>Median</td>
<td>Interquart. Range</td>
</tr>
<tr>
<td>Q4</td>
<td>ML1</td>
<td>4.0</td>
<td>2.0</td>
</tr>
<tr>
<td>ML2</td>
<td>4.0</td>
<td>1.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Q6</td>
<td>ML1</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>ML2</td>
<td>4.0</td>
<td>1.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Q7</td>
<td>ML1</td>
<td>4.0</td>
<td>1.0</td>
</tr>
<tr>
<td>ML2</td>
<td>4.0</td>
<td>1.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Key:

Both groups agreed that the Catalogue of Requirements, Design, and Code errors helped them understand error origin (Q4), cause (Q6), and prevention guidelines (Q7) (medians 4.0), with the exception being ML1 participants' agreement with Q6 which was only neutral with regard to the Catalogue of Design and Code Errors (medians 3.0). In all cases (Q4, Q6 and Q7), the medians of the ML2 distributions are better representatives than the medians of the ML1 distributions due to lower interquartile ranges (see table 7.8).

7.5.3.3 ML1 and ML2 Evaluation of Implementation of Error Diagnostic Perspectives
Table 7.9 summarises the ML1 and ML2 participants' evaluation about the way diagnostic perspectives were implemented in the Catalogue of Requirements, Design
and Code Errors. Both groups agree that the Catalogue of Errors helped them understand the symptoms (Q8) and triggers (Q9) of the errors documented in the Catalogue (medians 4.0) with the exception of the ML1 participants who were neutral with Q9 for the Catalogue of Requirements Errors (median 3.0).

Table 7.9 -- ML1 and ML2 Participants' Evaluation of Error Diagnostic Perspective in the Catalogue of Errors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q8</td>
<td>ML1 4.0 1.0</td>
<td>ML2 4.0 1.0</td>
<td></td>
</tr>
<tr>
<td>Q9</td>
<td>ML1 3.0 2.0</td>
<td>ML2 4.0 1.0</td>
<td></td>
</tr>
</tbody>
</table>

Key:
Q8: The Catalogue of Requirements/Design/Code Errors has helped to understand the symptoms of some requirements/design/code errors.
Q9: The Catalogue of Requirements/Design/Code Errors has helped me identify triggers to expose some requirements/design/code errors.

7.5.3.4 ML1 and ML2 Evaluation of Useability of the Catalogue of Errors

Table 7.10 summarises the data that was provided by the ML1 and ML2 participants to evaluate the useability of the three components of the Catalogue of Errors. The medians of the two groups of participants' evaluation for ease of use (Q1) of the Catalogue of Requirements, Design, and Code Errors are identical at 4, 3, and 3, respectively.

Table 7.10 -- ML1 and ML2 Participants' Evaluation of Useability of Catalogue of Errors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>ML1 4.0 2.0</td>
<td>ML2 3.0 2.0</td>
<td>ML2 3.0 2.0</td>
</tr>
<tr>
<td>Q2</td>
<td>ML1 2.0 2.0</td>
<td>ML2 3.0 2.0</td>
<td>ML2 3.0 2.0</td>
</tr>
</tbody>
</table>

Key:
Q1: The Catalogue of Requirements/Design/Code Errors is easy to use.
Q2: The Catalogue of Requirements/Design/Code Errors is easy to follow.
The ML1 participants disagreed that the Catalogue of Requirements Errors was easy to follow (Q2) as indicated by the median value of 2. However, they were neutral on their evaluation on Q2 for the Catalogue of Design and Code Errors (medians 3.0). The ML2 participants' evaluation shows agreement that the Catalogue of Requirements, and Code Errors was easy to follow (medians 4.0), but they were neutral about this evaluation for the Catalogue of Design Errors (median 3.0). In both cases (Q1 and Q2), the ML2 medians, represent their distributions better than ML1 medians represent theirs due to lower interquartile ranges (see table 7.10).

7.5.3.5 ML1 and ML2 Evaluation of Catalogue of Errors Usefulness to Identify and Correct Errors

Table 7.11 summarises the data provided by the ML1 and ML2 participants to evaluate whether the Catalogue of Requirements, Design and Code Errors helped them identify errors (Q10) and correct errors (Q11) in their software development artifacts. The median values suggest that ML1 participants were neutral in their evaluations of Q10 and Q11 (medians 3.0). The median values of ML2 indicate that these participants agreed with Q10 and Q11 (medians 4.0).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Q10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>ML2</td>
<td>4.0</td>
<td>1.0</td>
<td>4.0</td>
</tr>
<tr>
<td>Q11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>3.0</td>
<td>2.0</td>
<td>3.0</td>
</tr>
<tr>
<td>ML2</td>
<td>4.0</td>
<td>1.0</td>
<td>4.0</td>
</tr>
</tbody>
</table>

Key:
- Q10: The Catalogue of Requirements/Design/Code Errors has helped me identify some requirements/design/code errors.
- Q11: The Catalogue of Requirements/Design/Code Errors has helped me correct possible requirements/design/code errors.

7.5.3.6 ML1 and ML2 Evaluation of Overall Software Development Skill Improvement

Table 7.12 summarises ML1 and ML2 participants' evaluation of whether the components of the Catalogue of Errors helped in improving their overall skill level to

255
specify higher quality requirements, design and code artifacts (Q12). The median values show that the ML1 participants were neutral (medians 3.0), whereas the ML2 participants agreed with Q12 (medians 4.0).

Table 7.12 – ML1 and ML2 Participants’ Evaluation of Overall Skill Provided by the Catalogue of Errors

<table>
<thead>
<tr>
<th></th>
<th>ML1</th>
<th>Interquart. Range</th>
<th>ML2</th>
<th>Interquart. Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalogue of Requirements Errors</td>
<td>Median 3</td>
<td>Range 2</td>
<td>Median 4</td>
<td>Range 1</td>
</tr>
<tr>
<td>Catalogue of Design Errors</td>
<td>Median 3</td>
<td>Range 1</td>
<td>Median 4</td>
<td>Range 1</td>
</tr>
<tr>
<td>Catalogue of Code Errors</td>
<td>Median 3</td>
<td>Range 2</td>
<td>Median 4</td>
<td>Range 1</td>
</tr>
</tbody>
</table>

Key:
Q12: The Catalogue of Requirements/Design/Code Errors has improved my skills to specify higher quality requirements/design/code.

7.5.3.7 Summary and Evaluation

The examination of participant feedback to address the success factors (see tables 7.7 through to 7.12) of the Catalogue of Errors suggests that in general both ML1 and ML2 participants responses were above neutral on the 5-point scale agreement-disagreement questions. There were, however, some exceptions, which include the following situations:

i) Both ML1 and ML2 participants were neutral with the statements rating the Design and Code components of the Catalogue of Errors as easy to use (Q1). Both ML1 and ML2 participants were generally neutral on whether the Catalogue of Design Errors was easy to follow, however, only ML1 participants were neutral about this statement concerning the Catalogue of Code Errors. Also, the ML1 participants generally disagreed that the Catalogue of Requirements Errors was easy to follow.

ii) The ML1 participants indicated that they were neutral on whether the Catalogue Design and Code Errors had helped them understand about error severity (Q5). The ML2 participants were neutral on whether the Catalogue of Requirements Errors had helped them understand the severity of requirements errors.

iii) ML1 participants indicated that they were neutral on whether the Catalogue of Design and Code Errors had helped them understand error causes (Q6).
iv) The MLI participants also were neutral on whether the Catalogue of Requirement Errors had helped them identify triggers to expose the catalogued errors (Q9).

v) In general the MLI participants were neutral on whether the Catalogue of Errors had helped identify and correct errors in their artifacts.

vi) In general the MLI participants were neutral on whether the Catalogue of Errors had helped them improve their overall software development skills.

In order to understand the possible reasons why such feedback was provided the open-ended question responses were examined.

7.5.4 Open-ended Question Responses

The objective of this section is to present the feedback from the open-ended question of the questionnaires for the Catalogue of Requirements, Design, and Code Errors. The open-ended responses raised two broad categories of issues, namely, Recommendations (i.e. comments on further improvement) and Commendations (i.e. comments on strengths). Recommendations included issues that the participants felt were not properly addressed or implemented or could be improved further in the subsequent versions of the Catalogue. Commendations constituted good points or the strengths of the Catalogue. These two categories were broken down further into topic-based subcategories which were generalised after examining the open-ended anecdotal responses. A summary of the categories and subcategories is provided in table 7.13.

<table>
<thead>
<tr>
<th>Recommendations</th>
<th>Commentations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Clarification of terms</td>
<td>1) Catalogue of Errors is sufficiently detailed</td>
</tr>
<tr>
<td>2) More examples</td>
<td>2) Catalogue of Errors does not need further improvement</td>
</tr>
<tr>
<td>3) More time to be spent in class</td>
<td>3) Catalogue of Errors is useful to prevent errors</td>
</tr>
<tr>
<td>4) Improve layout/font</td>
<td>4) Catalogue of Errors has sufficient examples</td>
</tr>
<tr>
<td>5) More detail</td>
<td></td>
</tr>
<tr>
<td>6) Less detail</td>
<td></td>
</tr>
<tr>
<td>7) Include additional errors</td>
<td></td>
</tr>
<tr>
<td>8) Include syntax errors</td>
<td></td>
</tr>
<tr>
<td>9) Simplify error description</td>
<td></td>
</tr>
<tr>
<td>10) Provide online accessibility</td>
<td></td>
</tr>
</tbody>
</table>

Table 7.13 also highlights the most common (i.e. high occurrence) subcategories of comments provided. These and the associated frequencies for each component of the
Catalogue of Errors have been summarised in table 7.14. As table 7.14 shows, there were three principal aspects of the components of the Catalogue of Errors that attracted recommendations from a relatively large number of participants. Firstly, approximately 18.8%, 9.8%, and 4.8% of the ML1 participants indicated that the original version of the Catalogue of Requirements, Design and Code Errors, respectively, should have provided clarification of terminology in terms of glossaries, references, and explanations in lectures or workshops.

Table 7.14 – Summary of open-ended question comments and respective frequencies

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Clarification of terms</td>
<td>ML1 25 (n=133)</td>
<td>13 (n=133)</td>
<td>5 (n=126)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ML2 3 (n=67)</td>
<td>0 (n=67)</td>
<td>1 (n=63)</td>
<td></td>
</tr>
<tr>
<td>2. More Examples</td>
<td>ML1 30 (n=133)</td>
<td>26 (n=133)</td>
<td>12 (n=126)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ML2 6 (n=67)</td>
<td>10 (n=67)</td>
<td>8 (n=65)</td>
<td></td>
</tr>
<tr>
<td>3. More time to be spent in class</td>
<td>ML1 16 (n=133)</td>
<td>13 (n=133)</td>
<td>10 (n=126)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ML2 6 (n=67)</td>
<td>6 (n=67)</td>
<td>1 (n=65)</td>
<td></td>
</tr>
<tr>
<td>1. Catalogue of Errors is sufficiently detailed</td>
<td>ML1 12 (n=133)</td>
<td>2 (n=133)</td>
<td>0 (n=126)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ML2 4 (n=67)</td>
<td>0 (n=67)</td>
<td>2 (n=65)</td>
<td></td>
</tr>
<tr>
<td>2. No further improvement needed</td>
<td>ML1 4 (n=133)</td>
<td>4 (n=133)</td>
<td>0 (n=126)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ML2 0 (n=67)</td>
<td>0 (n=67)</td>
<td>0 (n=65)</td>
<td></td>
</tr>
<tr>
<td>3. Catalogue of Errors is useful to prevent errors</td>
<td>ML1 10 (n=133)</td>
<td>11 (n=133)</td>
<td>9 (n=126)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ML2 3 (n=67)</td>
<td>0 (n=67)</td>
<td>1 (n=65)</td>
<td></td>
</tr>
</tbody>
</table>

Comments with low occurrence have been not been included in table 7.14, however, they can be found in Appendix C, section 3.2.2.
Fewer ML2 participants, however, made similar comments for the revised version of the Catalogue of Errors (4.5%, 0.0%, and 1.5% for Catalogue of Requirements, Design and Code Errors, respectively). So this anomaly was addressed in the revised version of the Catalogue of Errors.

Another recommendation that recurred in the open-ended responses was that more examples were needed to illustrate the various types of errors in the Catalogue of Errors documentation. The proportions of the ML1 group who wanted this improvement constituted 22.6%, 19.6%, and 9.5%, respectively. However, there were fewer ML2 participants that expressed this recommendation for the revised version of the Catalogue of Errors. For example, 10.0%, 14.9%, and 12.3% of the ML2 participants wanted more examples in the Catalogue of Requirements, Design and Code Errors, respectively. It is believed that the differences in the above proportions are significant.

There was a relatively large number of requests from both ML1 and ML2 participants for more time to be spent on the Catalogue in lectures and workshops. Many indicated that this would have enhanced their understanding of the materials. The number of requests for additional time on the Catalogue of Requirements and Design Errors were comparable between ML1 (12.0% and 9.8%, respectively) and ML2 (10.0% and 10.0%, respectively). A notable difference, however, was observed between the number of requests made by ML1 (7.9%) and ML2 (1.5%) participants for additional time on the Catalogue of Code Errors.

As table 7.14 shows, some of the open-ended responses also highlighted commendations about the Catalogue of Errors. For example, some of the ML1 participants indicated that the Catalogues of Requirements and Design Errors were sufficiently detailed (9.0% and 1.5%, respectively). On the other hand, the same commendation was made by some ML2 participants for the Catalogues of Requirements and Code Errors (6.0% and 3.1%, respectively). 3% of the ML1 participants thought that no further improvement was required for both the Catalogue of Requirements and Design Errors. Also, non-negligible proportions of the ML1 participants felt they had to reiterate the fact that the three components of the Catalogues of Errors were indeed useful to prevent errors (7.5%, 8.3%, and 6.9%, respectively), while there were only
4.5% and 1.5% of the ML2 participants who felt they had to reiterate the usefulness of the Catalogue of Requirements and Code Errors to prevent errors.

Other recommendations made by fewer participants (see table 7.13) included the improvement of the layout and font of the Catalogue of Errors documentation, the simplification of error description and on-line accessibility. Some participants also indicated that the Catalogue of Errors should be enriched to include a wider variety of code errors, including syntax errors. There were also a few participants who thought that the Catalogue of Requirements and Design Errors needed no further improvement and that the Catalogue of Code Errors had sufficient examples (see appendix C, section 3.2.2 for frequencies).

7.5.5 The Feedback: Summary and Evaluation
In section 7.5, the feedback provided by the participants for the original and revised version of the Catalogue of Errors was examined. The feedback consisted of three elements, namely, time spent to learn a component of the Catalogue of Errors, participant responses to 5-point Likert scale agreement questions and open-ended question anecdotal responses. In this section, the three elements of feedback are linked in order to evaluate the success of the Catalogue of Errors.

In general, the feedback indicated that while the components of the Catalogue of Errors require some time to be assimilated, they did have an impact on the participants. In addition, the responses have also shown that the revised version of the Catalogue of Errors constituted an improvement on the original version.

The comparison between the ML1 and ML2 with respect to the three elements of feedback points to three observations:

i) The ML2 participants spent less of their own time to assimilate the revised version of the Catalogue than the ML1 participants did to assimilate the original version of the Catalogue. This means that the learnability of the revised version of the Catalogue of Errors was higher than in the original version. It is believed that the additional examples, the extra time spent, and the enhanced interactive approached, were all positive contributing factors which may need to be considered in future experimental designs.
ii) In general, the ML2 participants evaluated the revised version of the Catalogue better than the ML1 participants did evaluate the original version. This includes better responses with respect to the following success factors:
   a) Implementation of the Error Framework Perspectives in the Catalogue of Errors.
   b) Usability of the Catalogue of Errors.
   c) Usefulness of the Catalogue of Errors to Identify and Correct errors.
   d) Overall Software Development Skill Improvement

iii) In general, there were fewer ML2 participants making recommendations for the revised version of the Catalogue of Errors than ML1 participants who made recommendations for the original version.

These facts suggest that, in general, the feedback provided by the ML2 participants about the revised version of the Catalogue of Errors was more positive than that provided by the ML1 participants about the original version of the Catalogue of Errors. In both cases, ML1 and ML2 participants provided feedback with respect to the success factors outlined for the Catalogue of Errors (i.e., the outcome of research question two).

The improved feedback, however, does not suggest that there is no room for further improvement in the revised version of the Catalogue of Errors. It should be noted that none of the medians of the 5-point scale evaluation questionnaires for Q1-Q12 for the revised version of the Catalogue achieved a 5 (i.e., Strongly Agree) rating. In addition, there were still sizeable and non-negligible proportions of the ML2 participants who wanted further illustrative examples, additional clarifications and time to be spent in class. It is recommended that such issues be addressed in future trials of the Catalogue of Errors with more refined questionnaires and/or in-depth interviews with individual participants.

7.6 Catalogue of Errors: Potential Enhancements Identified

In the light of experience with the use of the Catalogue of Errors, some viable enhancements have been identified. These are outlined below:

i) Additional Errors. Increasing the number of documented errors can augment the content of the Catalogue of Errors. The Catalogue can include syntactic and non-
syntactic errors. The Catalogue can also include errors that are specific to specialised areas in Java, for example, JDBC errors, mathematical errors etc (Mitchell, 2000). Although, this is not part of the original objectives of this research the enhancement has already begun. Some non-syntactic Java errors have been identified and described (see appendix A, section 1.3). Future work requires that these errors (and others) be documented in accordance with the Error Framework described in chapter six and included in the Catalogue.

ii) Automation. Currently, the Catalogue of Errors can only be used manually. Work carried out to automate the Catalogue of Errors could enhance its usability considerably. For instance, a Catalogue of Errors Tool could be integrated with other software development artifact tools to automatically identify errors and present developers with error related information as artifacts are being developed.

iii) Unification of Prevention Guidelines. The existing versions of the Catalogue of Errors provide separate prevention guidelines for different errors. Such prevention guidelines are different and are applied in different phases of software development to different aspects of an artifact. One potential improvement could be to unify or integrate the prevention steps for different errors into a single error prevention framework.

7.7 Summary
The objective of this chapter was to describe the development of the Catalogue of Errors. It directly addressed research question two:

How can the Error Framework (developed in question one) be used to Catalogue Errors that are commonly injected in various software development artifacts?

In order to address this question a number of steps were followed. Initially the process that was adopted to develop the Catalogue of Errors was described, followed by the discussion of a number of content-related issues. Two versions of the Catalogue of Errors were produced, the original and the revised versions. The original version was used with ML1 participants who provided feedback on its contents and the mechanics used to introduce the Catalogue to them. The revised version of the Catalogue
incorporated some of the issues raised by the ML1 participants and was used with the ML2 participants who provided further feedback.

The chapter highlighted the differences in feedback from the original and the revised versions of the Catalogue of Errors. The changes in the revised version of the Catalogue of Errors were dictated by the feedback provided by ML1 participants about the original version. The revised version reused the content of its predecessor, but contained, additional examples and clarification of terminology in terms of additional references. Extra support (e.g. time spent in lectures and workshops) was also provided to the ML2 participants.

As a result of the revised version ML2 participants provided more positive feedback on the Catalogue of Errors, than the ML1 participants did about the original version. This difference is explained by the additional examples included in the revised version of the Catalogue and the additional support given to the ML2 participants. The chapter was ended with the identification of potential enhancements that can be made to Catalogue of Errors and pursued as future research directions.
8. Impact of the Catalogue of Errors

8.1 Overview

The objective of this chapter is to address research question three:

What is the impact of using of the Catalogue of Errors on software development?

In chapter three, this question was broken down into two more specific questions, namely,

iii) Does training of software developers with the Catalogue of Errors (developed by addressing question two) help reduce the number of errors injected by them into software development artifacts? If yes, can the reduction of the injected errors be quantified? "Reducing the number of errors" is defined to mean the following:
   c) preventing errors from being introduced into an artifact; or
   d) identifying and correcting errors injected in an artifact before the construction of subsequent artifacts starts.

iv) What is the effect of the use of the Catalogue of Errors on the productivity of software developers? Can this effect be quantified?

Note that in the two questions above the term quantify has been used. In both cases this term is defined to represent the measurement of the percentage increase or decrease in the following:

a) number of errors that are prevented from being introduced in an artifact,

b) number of errors that are removed from an artifact,

c) productivity of a developer to construct an artifact.

Clearly, this research question includes the evaluation of the impact of the Catalogue of Errors on the ability of developers to commit fewer errors, remove injected errors and their productivity in developing software artifacts. For this analysis, data were collected from participants who were involved in two field experiments (see section 4.2.1, chapter four). The objective of the collected data was to address the success factors of research.
question three. The collected data were analysed and the following conclusions were drawn.

Firstly, the Catalogue of Errors has a positive effect on the ability of developers to prevent errors from being introduced into a software development artifact. This effect is quantified and is found to be consistent with other studies reported in literature. However, unlike existing research, which has found a catalogue of C coding errors to be effective in preventing errors from being injected in C code artifacts (Yu, 1998), this study found that Catalogues of Requirements, Design, and Java code errors can be equally effective in preventing errors from being injected into all software development artifacts. Secondly, the Catalogue of Errors is found to have a positive impact on the ability of developers to remove errors in a software development artifact before the development of subsequent artifacts starts. The effect of the Catalogue of Errors on the ability to remove errors is quantified. Thirdly, in general, the Catalogue of Errors is found to have a negative impact on the productivity of developers to produce development artifacts. Such an effect is also quantified. The chapter concludes by arguing that the Catalogue of Errors should be used despite its adverse impact on productivity. These three findings are addressed in turn in the following sections.

8.2 Impact of Catalogue of Errors on Number of Injected Errors

The objective of this section is to address the first success factor for question three. In section 3.4.3, it was indicated that in order to determine the number of errors that are prevented from being injected into a development artifact, error density must be measured. A low error density for any artifact means that fewer errors are injected into an artifact or that more errors are prevented from being injected into an artifact. A high error density for any artifact means that more errors are injected into the artifact or that fewer errors are prevented from being injected into an artifact. Clearly, a low rather than high error density would constitute a successful outcome for research question three with respect to the first success factor, i.e. developer's ability to prevent errors (see section 3.4.3, chapter three).

In chapter four (see section 4.2.4.2, table 4.4) six hypotheses were defined in relation to the impact of the Catalogue of Errors on the number of errors that can be prevented by
the participants who took part in two field experiments. In order to achieve this, the error density of the development artifacts produced by ML1 and JO1 participants was measured. The six measurements have been summarised in table 8.1.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Formula Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density of Requirements Errors</td>
<td>Sum(Omissions, Ambiguities, Inconsistencies, Incorrect Facts) + Total Number of Pages of Requirements Artifact</td>
</tr>
<tr>
<td>Density of Other Requirements Errors</td>
<td>Sum(Other Requirements Errors (e.g. extraneous information etc.)) + Total Number of Pages of Requirements Artifact</td>
</tr>
<tr>
<td>Density of Design Errors</td>
<td>Sum(Algorithmic, Interface, Reuse, Strong Coupling, Weak Cohesion) + Total Number of Pages of Design Artifact</td>
</tr>
<tr>
<td>Density of Other Design Errors</td>
<td>Sum(Other Design Errors (e.g. class instance variables exposed, etc.)) + Total Number of Pages of Design Artifact</td>
</tr>
<tr>
<td>Density of Code Errors</td>
<td>Sum(Abuse of Namespaces, Class Invariants, Object Init, Confusing == with equals(), thread deadlocks) +Thousands Lines of Code (KLOC)</td>
</tr>
<tr>
<td>Density of Other Code Errors</td>
<td>Sum(Other code errors e.g. data type misuse, erroneous use of semicolons, braces, etc.) +Thousands Lines of Code (KLOC)</td>
</tr>
</tbody>
</table>

All the catalogued requirements and design errors in table 8.1 were committed by the participants. However, only five of the twelve catalogued code errors were included in the density computation for code errors. This is because none of the remaining seven code errors were committed by the participants of all groups. In addition, other requirements, design and code errors refer to errors that were not catalogued, but yet were committed by the participants in their artifacts. These were captured by the computation of the density of other requirements, design, and code errors formulae (see table 8.1).

Table 8.2 summarises the descriptive statistics (means and standard deviations) of the measurements that are shown in table 8.1. Figure 8.1 summarises the box plots of the density of requirements, design and code errors committed by the ML1, JO1, and ML2 participants. The box plots in figure 8.1 include some outliers and extreme values, which were also included in the analysis, because there was no indication that outliers and/or extreme values had occurred consistently for the same participant(s) in different artifacts due to non-treatment related reasons. Nevertheless, despite the effort to minimise these (see section 5.3.2, chapter five), it cannot be ruled out that one possible reason for the outliers and the extreme values may be due to maturation effects.
The data of table 8.2 and the patterns suggested by figure 8.1 will be examined more closely in the discussions that follow where the error density measurements of ML1, JO1, ML2 participants are compared.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>S. D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Density of Requirements Errors (DRE):</td>
<td>ML1</td>
<td>133</td>
<td>0.41</td>
<td>0.51</td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td>39</td>
<td>0.72</td>
<td>0.44</td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td>67</td>
<td>0.47</td>
<td>0.52</td>
</tr>
<tr>
<td>Density of Other Requirements Errors (DORE):</td>
<td>ML1</td>
<td>123</td>
<td>0.05</td>
<td>0.07</td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td>39</td>
<td>0.06</td>
<td>0.10</td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td>67</td>
<td>0.06</td>
<td>0.10</td>
</tr>
<tr>
<td>Density of Design Errors (DDE):</td>
<td>ML1</td>
<td>133</td>
<td>0.34</td>
<td>0.28</td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td>39</td>
<td>0.49</td>
<td>0.30</td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td>67</td>
<td>0.29</td>
<td>0.24</td>
</tr>
<tr>
<td>Density of Other Design Errors (DODE):</td>
<td>ML1</td>
<td>133</td>
<td>0.02</td>
<td>0.03</td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td>39</td>
<td>0.02</td>
<td>0.05</td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td>67</td>
<td>0.01</td>
<td>0.03</td>
</tr>
<tr>
<td>Density of Code Errors (DCE):</td>
<td>ML1</td>
<td>126</td>
<td>3.28</td>
<td>3.19</td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td>39</td>
<td>4.87</td>
<td>3.48</td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td>65</td>
<td>2.94</td>
<td>2.95</td>
</tr>
<tr>
<td>Density of Other Code Errors (DOCE):</td>
<td>ML1</td>
<td>126</td>
<td>0.61</td>
<td>0.92</td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td>39</td>
<td>0.55</td>
<td>0.76</td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td>65</td>
<td>0.52</td>
<td>0.77</td>
</tr>
</tbody>
</table>

8.2.1 Field Experiment One: ML1 versus JO1

Figure 8.1 (a through c) shows that the density of the catalogued requirements, design, and code errors that were injected by the ML1 participants are lower than those of JO1 participants. This is also confirmed by the means and standard deviations\(^4\) of such densities (see table 8.2), which are clearly different. For example, the mean density of requirements errors committed by the ML1 and JO1 participants were 0.41 and 0.72 (see table 8.2, figure 8.1, (a)), respectively.

It is important to determine whether the observed sample differences in error density of the various error types that are suggested by figure 8.1 (a through f) represent evidence that systematic differences exist between the population of those who would use the Catalogue of Errors and those who would not use the Catalogue of Errors.

\(^4\) The standard deviations of the various error density measurements have been included in order to show the variability of error densities of the ML1 and JO1 distributions. The logic behind the variability of a distribution (represented by standard deviation) is to give an idea of how far each observation in the distribution is from the mean (De Veaux, 2002).
Figure 8.1 - Box Plots of Distributions of Density of Catalogued and Other (Non-Catalogued) Errors
In order to achieve this, the hypotheses that were formulated in chapter four needed to be tested. The pairs of hypotheses concerning the density of errors have been reproduced in table 8.3. In addition, table 8.3 also includes the p-values that result from the application of the Mann-Whitney U significance test of the hypotheses on the error density data (see table 8.1 and 8.2).

| Table 8.3 – Mann-Whitney U test results for Density of Errors (ML1 vs JO1) |
|--------------------------|------------------|---|----------|
| No. | Null and Alternative Hypotheses | Group | N | p-value* |
| 1 | Density of Requirements Errors (DRE): $H_0$: DRE(ML1)=DRE(JO1) | ML1 | 133 | 0.001* |
|   | $H_1$: DRE(ML1)≠DRE(JO1) | JO1 | 39 | |
| 2 | Density of Other Requirements Errors (DORE): $H_0$: DORE(ML1)=DORE(JO1) | ML1 | 133 | 0.852 |
|   | $H_1$: DORE(ML1)≠DORE(JO1) | JO1 | 39 | |
| 3 | Density of Design Errors (DDE): $H_0$: DDE(ML1)=DDE(JO1) | ML1 | 133 | 0.002 |
|   | $H_1$: DDE(ML1)≠DDE(JO1) | JO1 | 39 | |
| 4 | Density of Other Design Errors (DODE): $H_0$: DODE(ML1)=DODE(JO1) | ML1 | 133 | 0.705 |
|   | $H_1$: DODE(ML1)≠DODE(JO1) | JO1 | 39 | |
| 5 | Density of Code Errors (DCE): $H_0$: DCE(ML1)=DCE(JO1) | ML1 | 126 | 0.002 |
|   | $H_1$: DCE(ML1)≠DCE(JO1) | JO1 | 39 | |
| 6 | Density of Other Code Errors (DOCE): $H_0$: DOCE(ML1)=DOCE(JO1) | ML1 | 126 | 0.827 |
|   | $H_1$: DOCE(ML1)≠DOCE(JO1) | JO1 | 39 | |

*p<0.05 – Distributions have significantly different locations;

In the following sections the hypotheses concerning the density of catalogued and non-catalogued errors are examined.

Density of Catalogued Errors

The null hypotheses 1, 3, and 5 (see table 8.3) assert that the ML1 participants' density of the requirements, design and code errors is the same as that of JO1 participants respectively. These hypotheses are rejected. First, the ML1 participants committed an average of 75.6% fewer catalogued requirements errors in their requirements artifacts as opposed to their JO1 counterparts (means: 0.41 versus 0.72 (see table 8.2)).
The result of the Mann-Whitney U test for hypothesis 1 (p = 0.001 < 0.05, see table 8.3) suggests that there is not enough evidence to support the null hypothesis, and, therefore, the alternative hypothesis (Hₐ: Density of Requirements Errors (ML₁) ≠ Density of Requirements Errors (JO₁)) is supported.

Second, the ML₁ participants committed an average of 44.1% fewer catalogued design errors in their design artifacts as opposed to their JO₁ counterparts (means: 0.34 versus 0.49 (see table 8.2)). The result of the Mann-Whitney U test for hypothesis 3 (p = 0.002 < 0.05, see table 8.3) suggests that there is not enough evidence to support the null hypothesis, and, therefore, the alternative hypothesis Hₐ: Density of Design Errors (ML₁) ≠ Density of Design Errors (JO₁) is supported.

Third, the ML₁ participants committed an average of 48.5% fewer catalogued code errors in their code artifacts as opposed to their JO₁ counterparts (means: 3.28 versus 4.87 (see table 8.2)). The result of the Mann-Whitney U test for hypothesis 5 (p = 0.002 < 0.05, see table 8.3) suggests that there is not enough evidence to support the null hypothesis, and, therefore, the alternative hypothesis Hₐ: Density of Code Errors (ML₁) ≠ Density of Code Errors (JO₁) is supported.

On the basis of the results of the significance tests shown above it can be said that there is a difference in the densities of requirements, design and code errors between the ML₁ and the JO₁ groups. These results are consistent with Yu (1998) where the number of code errors that were prevented after the catalogue of C code errors was introduced to developers was 34.5%. The difference between the results reported here and Yu's (48.5% versus 34.5%) may be due to the different size, complexity, application domain of the applications from which the data were collected.

It is worth noting that the number of the prevented requirements errors in this study (75.6%) is notably higher than the number of the prevented design and code errors (44.1% and 48.5%, respectively). This result prompted the examination of the data collected via the Software Development Background Questionnaire (see appendix C,
section 3.1.3) for the ML1 participants. This data includes participant perceived proficiency in requirements specification, high- and low-level design and code indications. It was suspected that maybe the ML1 participants indicated that they were more proficient at specifying requirements, than specifying high-, low-level designs, or code. The comparison of the descriptive statistics (median, range, percentiles) reveals similarity between ML1 participants’ perceived proficiency in requirements, high- and low-level design and code (see appendix C, section 3.1.3). This suggests that there probably is another reason why more requirements errors were prevented than design and code errors.

The fact that more requirements errors were prevented as opposed to design and code errors may be explained by the fact that the prevention guidelines of the Catalogue of Requirements Errors are different from the prevention guidelines of the Catalogue of Design and Code errors. The former, in addition to disseminating knowledge about requirements errors includes suggestions about how to actually construct a requirements artifact (e.g. goal, viewpoint, and scenario generation and analysis). The Catalogues of Design and Code Errors, on the other hand, simply focus on what can be done to prevent selected errors, without suggesting any particular way to construct design and code artifacts (see section 7.3.5.4, chapter seven, and appendix A, sections 1.1 and 1.2). In addition, the Catalogue of Requirements Errors requires the construction of traceability matrices, which force the participants/developers to review the requirements outlined in their requirements artifact. This increases the chance that even if requirements errors are injected into requirements artifacts, they are more likely to be removed when the traceability matrices are constructed. This implies that future upgrades of the Catalogue of Design and Code Errors can be enhanced by including artifact construction guidelines as well.

**Density of Other Errors**

Figure 8.1 (d through f) shows the density of other (i.e. non-catalogued) requirements, design and code errors that was committed by ML1 participants is comparable to that of JO1 participants. This is supported by the data of table 8.2 as well. For example, the
means of the density of other requirements errors for both ML1 and JO1 participants were 0.05 and 0.06 respectively (see table 8.2).

Null hypotheses 2, 4, and 6 (table 8.3) assert that the density of non-catalogued requirements, design and code errors for ML1 participants will be the same as JO1 participants’ respectively. None of these hypotheses can be rejected, because there is not enough evidence to indicate that the alternative hypotheses are true (Mann-Whitney U tests, \( p > 0.05 \) (see table 8.3)). This suggests that non-catalogued errors were injected in comparable amounts by both ML1 and JO1 participants. This result was expected because, all participants had the same knowledge of non-catalogued requirements, design and code errors.

### 8.2.2 Field Experiment Two: ML1 versus ML2

Figure 8.1 (a through f) also includes box plots of the distributions of the density of catalogued (requirements, design and code) and other non-catalogued errors committed by the ML2 participants. The box plots suggest that ML2 error density distributions for the catalogued errors are consistent with ML1 and different from JO1. This is confirmed by the descriptive statistics (i.e. means and standard deviations) of the densities of errors, which have been included in table 8.2. For example, the means of the density of requirements errors for ML1 and ML2 participants were 0.41 and 0.47, respectively. The density of requirements errors for JO1 participants is clearly higher at 0.72 (see table 8.2). A similar pattern exists with the density of design errors (Density\(_{ML1}\) = 0.34; Density\(_{JO1}\) = 0.49; Density\(_{ML2}\) = 0.29) and code errors (Density\(_{ML1}\) = 3.28; Density\(_{JO1}\) = 4.87; Density\(_{ML2}\) = 2.94) (see table 8.2).

The density of requirements, design and code errors of ML2 participants were not included in the inferential tests of significance because the ML1 and ML2 participants attempted two different projects of similar complexity, yet at different times. Nevertheless, the similarity between the ML1 and ML2 error density distributions and descriptive statistics (see figure 8.1 and table 8.2) lends support to the conclusion that the Catalogue of Errors did indeed have a positive impact on the developers who use it to help them commit fewer errors. In addition, the ML2 error density data increase confidence in the results in the first field experiment (ML1 versus JO1).
As a consequence, the similarity between the ML1 and ML2 error density data is only demonstrated using descriptive statistics. Despite this, additional replications of the field experiments are encouraged.

8.2.3 Impact of Catalogue of Errors on Number of Injected Errors: Summary
In the previous sections, the results obtained by examining the density of catalogued errors and other non-catalogued errors confirm previous findings (e.g. Yu, 1998)) that knowledge about errors can indeed help developers commit fewer errors. However, while Yu (1998) focused on code errors only, this section has shown that Catalogues of Requirements and Design Errors can help developers commit significantly fewer requirements and design errors as well. Furthermore, the coding errors were Java rather than C errors, showing that a Catalogue of Errors can work with different languages. It was also shown that the Catalogue of Requirements Errors was more effective than the Catalogue of Design and Code Errors. This was shown in the results of the first field experiment with the original version of the Catalogue of Errors in semester 1 2001 (i.e. ML1 versus JO1) and also it was supported by the partial internal replication in the second field experiment in semester 2 2001 (i.e ML1 versus ML2) with the revised version of the Catalogue of Errors. Therefore, the results show a successful outcome of research question three with regard to the first success factor. Taken together, the results from both field experiments (ML1 versus JO1 and ML2) suggest that the generalisability of the results of the field experiments to the population of interest is not unreasonable.

The analysis of the data also suggests a limitation inherent in the decision of what errors to catalogue (see section 7.3.1). While such a decision was meant to make the Catalogue of Errors as comprehensive as possible by using experience and by surveying the literature about possible requirements, design and code errors, it also ran the risk of losing the relevance for the participants. This is observed in the computation of Density of Code Errors (see table 8.1), where only five of the twelve catalogued code errors were actually committed by the ML1, JO1, and ML2 participants (see section 8.2). Consequently, it is recommended that future replications of the field studies and experiments, should be preceded by pilot studies (if there is certainty they will not
compromise experiment results with a maturation effect to determine what errors need to be catalogued before actually building a catalogue and demonstrating its usefulness.

8.3 Impact of Catalogue of Errors on Error Correction Ability: Escape Ratios

This section compares and contrasts the error correction ability of the ML1, JO1 and ML2 participants. Table 8.5 summarises the descriptive statistics of the distributions of errors that escaped (i.e., were not corrected in the development artifact in which they were originated) by the participants for the various groups. The first row of the table summarises the data representing the descriptive statistics of requirements errors that were found in design and code artifacts. The second row of the table summarises the data representing the descriptive statistics of design errors that were found in code artifacts.

<table>
<thead>
<tr>
<th>Catalogued Errors Injected</th>
<th>Group</th>
<th>Catalogued Errors Found</th>
<th>Code Artifact</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Design Artifact</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>S. D.</td>
</tr>
<tr>
<td>Requirements Artifact</td>
<td>ML1</td>
<td>1.02</td>
<td>1.41</td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td>2.67</td>
<td>1.42</td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td>0.91</td>
<td>1.14</td>
</tr>
<tr>
<td>Design Artifact</td>
<td>ML1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>JO1</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ML2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From table 8.5 it appears that JO1 participants left more errors uncorrected than their ML1 and ML2 counterparts. For example, the mean number of design errors escaping to code artifacts for the ML1, ML2 and JO1 participants were 1.43, 2.82, and 1.63 respectively (see table 8.5). A similar pattern exists for the mean number of requirements errors escaping to design and code artifacts (see table 8.5).

As shown in chapter three, error correction ability can be determined by measuring the escape ratios, i.e., errors that are not identified and corrected in the artifact where they were originated. Comparing developers on the basis of the number of errors that they failed to identify and correct in a development artifact, allowing them to be carried forward to subsequent development artifacts would help distinguish between developers.
who have an awareness about errors and the ability to remove them and those who do not.

Escape ratios are defined as follows: If $a$ represents a software development artifact and $a'$ represents another software development artifact which is developed immediately after $a$; and if $e$ represents the number of errors that are introduced in $a$; and if $e'$ is the number of errors of artifact $a$ that still remain in artifact $a'$ after $a'$ has been constructed, the ratio between $e'/e$ represents the escape ratio of the errors that have escaped correction in artifact $a$ and are still present in artifact $a'$. Escape ratios can be defined for different artifacts.

As shown in section 4.2.4.2 (see chapter four), three definitions of escape ratios were measured, namely, the Escape Ratios of Requirements Errors Escaping to Design and Code Artifacts and the Escape Ratio of Design Errors Escaping to Code Artifacts. The formulae to compute the above escape ratios are summarised in table 8.6.

<table>
<thead>
<tr>
<th>Measurement</th>
<th>Formula Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escape Ratio of Requirements Errors Escaping to</td>
<td>$\frac{\text{Sum(omissions, ambiguities, inconsistencies, incorrect facts) found in Design Artifact}}{\text{Sum(omissions, ambiguities, inconsistencies, incorrect facts) found in Requirements Artifact}}$</td>
</tr>
<tr>
<td>Design Artifacts</td>
<td></td>
</tr>
<tr>
<td>Escape Ratio of Requirements Errors Escaping to</td>
<td>$\frac{\text{Sum(omissions, ambiguities, inconsistencies, incorrect facts) found in Design Artifact}}{\text{Sum(omissions, ambiguities, inconsistencies, incorrect facts) found in Requirements Artifact}}$</td>
</tr>
<tr>
<td>Code Artifacts</td>
<td></td>
</tr>
<tr>
<td>Escape Ratio of Design Errors Escaping to Code</td>
<td>$\frac{\text{Sum(algorithmic, interface, reuse, strong coupling and weak cohesion errors) found in Code Artifact}}{\text{Sum(algorithmic, interface, reuse, strong coupling and weak cohesion errors) found in Code Artifact}}$</td>
</tr>
<tr>
<td>Artifacts</td>
<td></td>
</tr>
</tbody>
</table>

Normally, the escape ratios shown in table 8.6 are expected to range from 0 to 1. An escape ratio value closer to 0 means that fewer errors have escaped to subsequent artifacts than an escape ratio value closer to 1. This means that a low rather than a high escape ratio value would constitute a successful outcome for question three with regard to the second success factor, i.e. ability of developers to remove errors from the artifact of origin (see section 3.4.3, chapter three)
It is recognised that it is also possible for escape ratios to assume values greater than 1. An example of such a case would be when a developer introduces additional requirements errors in a design artifact (e.g. a correct requirement that was originally provided for in the requirements artifact is not designed in the design artifact).

The escape ratio measurements that are shown in table 8.6 compare the ML1 and JO1 participants. Figure 8.2 (a through c) shows box plots representing the distributions of these measurements. For comparison purposes the distributions of the ML2 participants are included as well. Table 8.7 summarises the descriptive statistics (means and standard deviations) of the escape ratio measurements for the ML1, JO1, and ML2 participants.

<table>
<thead>
<tr>
<th>Group</th>
<th>N</th>
<th>Mean</th>
<th>S. D.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Escape Ratio of Requirements Errors Escaping to Design Artifacts (PREDMA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>112</td>
<td>0.37</td>
<td>0.40</td>
</tr>
<tr>
<td>JO1</td>
<td>37</td>
<td>0.63</td>
<td>0.32</td>
</tr>
<tr>
<td>ML2</td>
<td>47</td>
<td>0.38</td>
<td>0.34</td>
</tr>
<tr>
<td>Escape Ratio of of Requirements Errors Escaping to Code Artifacts (PREDCA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>103</td>
<td>0.11</td>
<td>0.23</td>
</tr>
<tr>
<td>JO1</td>
<td>37</td>
<td>0.17</td>
<td>0.19</td>
</tr>
<tr>
<td>ML2</td>
<td>47</td>
<td>0.10</td>
<td>0.17</td>
</tr>
<tr>
<td>Escape Ratio of of Design Errors Escaping to Code Artifacts (PDCA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>103</td>
<td>0.35</td>
<td>0.56</td>
</tr>
<tr>
<td>JO1</td>
<td>36</td>
<td>0.65</td>
<td>0.32</td>
</tr>
<tr>
<td>ML2</td>
<td>60</td>
<td>0.38</td>
<td>0.34</td>
</tr>
</tbody>
</table>

8.3.1 Escape Ratios: ML1 versus JO1

Figure 8.2 (a through c) shows a difference between the ML1 and JO1 distributions with respect to the escape ratios. The escape ratios of the ML1 participants are lower than those of the JO1 participants. This is also confirmed in table 8.7 which shows that the means of the escape ratios of requirements errors escaping to design (ML1= 0.37; JO1= 0.63; ML2= 0.38) and code artifacts (ML1= 0.11; JO1= 0.17; ML2= 0.19) and those of design errors escaping to code artifacts (ML1= 0.55; JO1= 0.65; ML2= 0.58) are lower for both the ML1 and ML2 participants when compared to the JO1 participants.

Note that the value of N (size of the sample) for the escape ratio distributions are different from the sample size of the number of participants in ML1, JO1 and ML2. This is because the escape ratio measurements are ratios. In all measurements the denominator of such ratios constitutes the number of errors that are originally found in an artifact. Many participants produced artifacts where no errors were found. In such cases the denominators of the escape ratios for some participants contributed 'division by zero' and therefore, the values of escape ratios could not be computed. Consequently, sample sizes for the escape ratio distributions were smaller.
Figure 8.2 – Box Plots of Distribution of Escapes Ratios

- a -

- b -

- c -
In order to determine whether the observed differences in the escape ratios between ML1 and JO1 occur systematically in the populations of students, it is important to test the hypotheses that were formulated in section 4.2.4.2 in table 4.4 (see chapter four) and which are reproduced in table 8.8 for convenience.

Table 8.8 also contains the p-values that result from the Mann-Whitney U test of the hypotheses.

<table>
<thead>
<tr>
<th>No</th>
<th>Null and Alternative Hypotheses</th>
<th>Group</th>
<th>N</th>
<th>p-value</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Escape Ratio of Requirements Errors Escaping to Design Artifacts (ERREDA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$H_0$: ERREDA (ML1) = ERREDA (JO1)</td>
<td>ML1</td>
<td>112</td>
<td>0.001**</td>
</tr>
<tr>
<td></td>
<td>$H_1$: ERREDA (ML1) $&lt;$ ERREDA (JO1)</td>
<td>JO1</td>
<td>27</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Escape Ratio of Requirements Errors Escaping to Code Artifacts (ERRECA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$H_0$: ERRECA (ML1) = ERRECA (JO1)</td>
<td>ML1</td>
<td>105</td>
<td>0.002</td>
</tr>
<tr>
<td></td>
<td>$H_1$: ERRECA (ML1) $&gt;$ ERRECA (JO1)</td>
<td>JO1</td>
<td>37</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Escape Ratio of Design Errors Escaping to Code Artifacts (EREDCA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$H_0$: EREDCA (ML1) = EREDCA (JO1)</td>
<td>ML1</td>
<td>108</td>
<td>0.008</td>
</tr>
<tr>
<td></td>
<td>$H_1$: EREDCA (ML1) $&lt;$ EREDCA (JO1)</td>
<td>JO1</td>
<td>35</td>
<td></td>
</tr>
</tbody>
</table>

*p<0.05 — Distributions have significantly different locations;

The null hypotheses 7, 8, and 9 in table 8.8 assert that the ML1 participants’ escape ratios are the same as those of JO1 participants. These hypotheses are rejected. First, the ML1 participants allowed 79.3% (means: 0.37 versus 0.63 (see table 8.7)) fewer requirements errors to escape to design artifacts as opposed to their JO1 counterparts. The result of the Mann-Whitney U test for hypothesis 7 (p = 0.001<0.05, see table 8.8) suggests that there is not enough evidence to support the null hypothesis. Consequently, the alternative hypothesis that the Escape Ratios of Requirements Errors Escaping to Design Artifacts of ML1 participants is significantly different from the Escape Ratios of Requirements Errors Escaping to Design Artifacts of JO1 participants is supported.

Second, the ML1 participants allowed 54.5% (means: 0.11 versus 0.17 (see table 8.7)) fewer requirements errors to escape to code artifacts as opposed to their JO1

**The p-value generated by the statistical processing was 0.000. Reporting such value would be statistically incorrect and it was therefore decided to use 0.001 instead to represent a very small probability.
counterparts. The result of the Mann-Whitney U test for hypothesis 8 (p = 0.002<0.05, see table 8.8) suggests that there is not enough evidence to support the null hypothesis. Consequently, the alternative hypothesis that the Escape Ratios of Requirements Errors Escaping to Code Artifacts of ML1 participants is significantly different from the Escape Ratios of Requirements Errors Escaping to Code Artifacts of JO1 participants is supported.

Third, the ML1 participants allowed 18.2% (means: 0.55 versus 0.65 (see table 8.7)) fewer design errors to escape to code artifacts as opposed to their JO1 counterparts. The result of the Mann-Whitney U test for hypothesis 9 (p = 0.008<0.05, see table 8.8) suggests that there is not enough evidence to support the null hypothesis. As a result, the alternative hypothesis that the Escape Ratios of Design Errors Escaping to Code Artifacts of ML1 participants is significantly different from the Escape Ratios of Design Errors Escaping to Code Artifacts of JO1 participants is supported.

The above results suggest that the ML1 participants identified and corrected notably larger numbers of requirements errors in the actual requirements artifacts (70.3% and 54.5%) in comparison to their JO1 counterparts. This is considered as a successful outcome because the presence of requirements errors constitutes the main reason why many projects fail in the industry (Davis, 1993a; Lamsweerde, 2000; Pressman, 1997). In addition, the benefit of identifying and correcting requirements error sooner rather than later constitutes additional cost savings, because the later in the development requirements errors are identified the more expensive their correction and removal (Heitmeyer et al., 1996; Hevner, 1997).

Despite this successful outcome it should also be noted that the number of design errors that were identified and corrected by ML1 participants as opposed to JO1 participants is notably lower than the requirements errors (70.3% and 54.5% versus 18.2%). The investigation of the raw data of the design errors that were originally found in design artifacts and later in code artifacts (see appendix C, section 3.3.4) suggests that the design errors that escaped to code artifacts were mainly strong coupling and weak cohesion errors. One possible reason for this is the fact that the ML1 participants found such errors harder to correct in comparison to other design errors. Another reason could also be that typically such errors do not affect the functionality of the code, nor do they
affect the requirements delivered through the code. Strong coupling and weak cohesion errors affect the future reusability and modifiability of the code (see appendix A, sections 1.1 and 1.2).

The participants appear to have focused on producing working code rather than code that can be easily reused and modified. This behaviour of student participants is consistent with McAndrews' (2000) view, and therefore, not totally surprising:

"Students [participants] do whatever it takes to write programs, compile them and turn them in. They seldom worry about disciplined methods and quality. Nor are they overly concerned with planning their assignments, with the exception that they probably want to get them done as quickly as possible." (p. 6).

This also suggests a limitation of this research, and is directly related to the fact that student participants were used. Consequently, future replications using student participants should stress greater reusability and modifiability (and other quality aspects) in the code artifacts.

8.3.2 Escape Ratios ML1 versus ML2

The escape ratio data for the ML2 participants have been included table 8.7 and figure 8.2 (a through c) along with ML1 and JO1 escape ratio data. The data suggests that the distributions of escape ratios for the ML2 participants were closer to the distributions of the ML1 participants than they were to the JO1 participants. For example, the means of the escape ratio of the requirements errors found in the design artifacts for ML1 and ML2 were 0.37 and 0.38, respectively. The mean of the requirements errors escaping to design artifacts for JO1 participants was 0.63. The same thing can be said about the means of the requirements and design errors escaping to code artifacts (see table 8.7).

The escape ratio data of ML1 and ML2 participants was not included in the inferential tests of significance because the ML1 and ML2 participants attempted two different projects of similar complexity, at different times. Nevertheless, it should be said that the descriptive statistics support the impact that the Catalogue Errors had on the ability of the ML2 participants to correct more errors in the artifact of origin rather than in subsequent artifacts. This finding also increases the confidence in the results obtained in
the field experiment in semester 1, 2001, where MLI escape ratios were compared with JOI escape ratios.

8.3.3 Impact of Catalogue of Errors on Error Correction Ability: Summary
The previous sections have shown that the Catalogue of Errors had an impact in helping developers remove errors from artifacts early in software development rather than later. This was done by measuring the errors that escape the artifact of origin into subsequent artifacts for the MLI, JOI, and the ML2 participants. The analysis of the data has shown that the MLI participants who used the Catalogue of Errors managed to allow fewer requirements and design errors to escape artifacts of origin as opposed to the JOI participants who did not use the Catalogue of Errors. In addition, it was also shown that the number of requirements errors that were corrected sooner rather than later was notably larger than the number of design errors. While ML2 participant data were not conclusive, there was support for the conclusions made for MLI and JOI participants. It is not therefore unreasonable to generalise these findings to the population from which the samples were drawn. In summary, the outcome of research question three with regard to the ability of developers to remove errors from the artifacts of origin (i.e. second success factor, see section 3.4.3, chapter four) is successful.

8.4 Impact of Catalogue of Errors on Productivity
In this section the impact of the Catalogue of Errors on the productivity of the participating groups (i.e. MLI, JOI and ML2) is addressed. In order to achieve this, the productivity of the participants to produce requirements, design, and code artifacts was measured. These measurements were initially introduced and defined in section 3.4.3 (see chapter three). A summary has been reproduced in table 8.9.

| Table 8.9 – Productivity to Develop Requirements, Design and Code Artifacts |
|-----------------|-----------------|-----------------|
| Measurements | Formula Definition | Definitions |
| Productivity to develop Requirements Artifact | Number of Pages of Requirements Artifact + Time spent to develop Requirements Artifact (in hours) | |
| Productivity to develop Design Artifact | Number of Pages of Design Artifact + Time spent to develop Design Artifact (in hours) | |
| Productivity to develop Code Artifact | Number of Thousands of Lines of Code (KLOC) in Code Artifacts + Time spent to develop Code Artifact | |
Higher values of productivity to produce an artifact are better than lower values. This is because higher values mean that developers are producing more output for the same amount of time spent on the construction of the artifact. Consequently, a successful outcome of research question three will be high developer productivity.

The distributions of the productivity measurements of table 8.9 are portrayed using box plots for ML1, JO1 and ML2 participants in figure 8.3 (a through c). Table 8.10 summarises the descriptive statistics (means and standard deviations) for these distributions.

| Table 8.10 – Descriptive Statistics for Productivity to Develop Artifacts (ML1 vs JO1 vs ML2) |
|-------------------------------------------------|-----------------|----------|----------|
| Productivity to develop Requirements Artifact  | Group | N | Mean | S. D. |
| (PDRA):                                         | ML1 | 133 | 0.90 | 0.39 |
|                                                | JO1 | 39  | 1.00 | 0.34 |
|                                                | ML2 | 67  | 0.77 | 0.20 |
| Productivity to Develop Design Artifact (PDDA): | ML1 | 133 | 0.83 | 0.28 |
|                                                | JO1 | 39  | 0.80 | 0.20 |
|                                                | ML2 | 67  | 0.96 | 0.22 |
| Productivity to Develop Code Artifact (PDCA):   | ML1 | 126 | 38.65 | 16.67 |
|                                                | JO1 | 39  | 47.20 | 22.15 |
|                                                | ML2 | 65  | 43.28 | 17.92 |

The following sections compare the productivity measurements made during the first field experiment (semester 1, 2001) and its partial internal replication (semester 2, 2001).

8.4.1 Field Experiment One: ML1 versus JO1

Figure 8.3 (a through c) shows that the distribution of the productivity of ML1 and JO1 participants to produce requirements and code artifacts varies. The JO1 participants appear to have a slightly higher productivity when constructing requirements and code artifacts than their ML1 counterparts (figure 8.3, a and c). This is also confirmed by the descriptive statistics summarized in table 8.10. The distributions of productivity when developing design artifacts for ML1 and JO1, however, do not appear to vary as much.
Figure 8.3 - Box Plots of Productivity to Develop Requirements, Design and Code Artifacts
In order to test the significance of the observed difference between the productivity of ML1 and JO1 participants and to determine whether the observed differences are indeed evidence of systematic differences in the productivity of the populations from which the samples were drawn, the hypotheses about productivity that were formulated in chapter four (see section 4.2.4.2, table 4.5) need to be tested.

For convenience, these hypotheses have been reproduced in table 8.11. In addition, table 8.11 also contains the p-values that result from the application of Mann-Whitney U significance test on such hypotheses.

Table 8.11 - Mann-Whitney U test results for Productivity to Develop Artifacts (ML1 vs JO1)

<table>
<thead>
<tr>
<th>No</th>
<th>Null and Alternative Hypotheses</th>
<th>Group</th>
<th>N</th>
<th>p-value*</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>Productivity to Develop Requirements Artifacts (PDRA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H0: PDRA (ML1) = PDRA (JO1)</td>
<td>ML1</td>
<td>133</td>
<td>0.015</td>
</tr>
<tr>
<td></td>
<td>H1: PDRA (ML1) ≠ PDRA (JO1)</td>
<td>JO1</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Productivity to Develop Design Artifact (PDDA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H0: PDDA (ML1) = PDDA (JO1)</td>
<td>ML1</td>
<td>133</td>
<td>0.821</td>
</tr>
<tr>
<td></td>
<td>H1: PDDA (ML1) ≠ PDDA (JO1)</td>
<td>JO1</td>
<td>39</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>Productivity to Develop Code Artifact (PDCA):</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>H0: PDCA (ML1) = PDCA (JO1)</td>
<td>ML1</td>
<td>126</td>
<td>0.032</td>
</tr>
<tr>
<td></td>
<td>H1: PDCA (ML1) ≠ PDCA (JO1)</td>
<td>JO1</td>
<td>39</td>
<td></td>
</tr>
</tbody>
</table>

* p<0.05 - Distributions have significantly different locations;

The null hypotheses 10, 11, and 12 assert that the productivity of the ML1 participants to produce requirements, design and code artifacts is the same as that of JO1 participants. The p-values in table 8.11 suggest three conclusions. First, the null hypothesis 10 must be rejected in favour of the alternative hypothesis (Mann-Whitney U test p=0.015<0.05, see table 8.11). On average, the productivity of the ML1 participants when constructing requirements artifacts was 11.1% lower (means: 0.90 versus 1.00 (see table 8.10)) than the productivity of JO1 participants when constructing requirements artifacts. Hence, a possible explanation is that the Catalogue of Requirements Errors has adversely affected the productivity of the ML1 participants.

Second, the null hypothesis 11 cannot be rejected because there is not enough evidence to indicate that the alternative hypothesis is true (Mann-Whitney U test p = 0.821 > 0.05, see table 8.11). This suggests that the distributions of productivity when developing design artifacts for the ML1 and JO1 participants do not have significantly
different locations. This is despite the fact that on average it appears as though the ML1 participants' productivity when constructing design artifacts is 3.8% higher (means: 0.83 versus 0.80 (see table 8.10)) than the productivity of JO1 participants. Hence, the Catalogue of Design Errors may not have had a significant impact on the productivity of the ML1 participants when producing design artifacts.

Third, null hypothesis 12 must be rejected in favor of the alternative hypothesis (Mann-Whitney U test p = 0.032 < 0.05, see table 8.11). On average the productivity of the ML1 participants when producing code artifacts was 22.1% lower (means: 38.65 versus 47.20 (see table 8.10)) than the productivity of the JO1 participants. Hence, the Catalogue of Code Errors had a significant negative impact on the productivity of the ML1 participants to produce code artifacts as opposed to the JO1 participants.

While a decreased productivity among ML1 participants when producing requirements and code artifacts is not desirable, it was not totally unexpected. As suggested earlier, the ML1 participants who used the Catalogue of Requirements Errors were supposed to review their requirements using traceability matrices, etc., which were not used by the JO1 participants. Such additional activities were expected to take additional time. Similarly, the Catalogue of Code Errors itself required the addition of extra code into the simulator (e.g. additional methods in classes, additional object initialisation code, etc. see appendix A, sections 1.1 and 1.2). The JO1 participants, who did not get the Catalogue of Errors documentation, were not expected to add such additional code.

The fact that the productivity of the ML1 participants to develop design artifacts is not significantly different from that of JO1 participants is not surprising when the content of the Catalogue of Design Errors is examined. One possible reason for this is the nature of the prevention guidelines associated with the various design errors. Algorithmic, reuse and interface errors are the only ones that require developers to undertake certain steps to prevent such errors. The other errors, however, (e.g. strong coupling and weak cohesion) do not require developers to perform any actions besides acquiring the knowledge of the various acceptable and unacceptable levels of coupling and cohesion. The time spent to acquire knowledge about such coupling and cohesion levels would have been indicated in the Catalogue of Errors Learning time (see section 7.5.2, chapter seven).
8.4.2 Field Experiment Two: MLI versus ML2

The comparison of MLI and ML2 descriptive statistics (means and standard deviation) of the productivity when developing requirements, design and code artifacts (see table 8.10), and the box plots depicting such distributions (see figure 8.3 (a through c)) show a similarity.

When viewed together, it appears that the MLI productivity data is more similar to the ML2 productivity data than to the JO1 data. While such similarity cannot be considered conclusive, it does lend support to the view that the MLI and ML2 participants had similar patterns of productivity when constructing artifacts. In both cases, productivity was adversely affected and this is due to the fact that MLI and ML2 participants both used the Catalogue of Errors. Despite this outcome, additional replications of these field experiments to study productivity data further are encouraged.

8.4.3 Impact of Catalogue of Errors on Productivity: Summary

Section 8.3 has compared the productivity of MLI and JO1 participants when developing requirements, design and code artifacts. It was shown that, due to the introduction of the Catalogue of Errors, the productivity of MLI participants to develop requirements and code artifacts was actually lower than that of JO1 participants for the same artifacts. It was also shown that the productivity of MLI and JO1 participants when developing design artifacts was not significantly different. The examination of the ML2 data, although inconclusive, supports the fact that the Catalogue of Errors had an adverse impact on the productivity of participants. Therefore, the outcome of research question three with regard to the ability of software developers to be productive in their work (i.e. research question three success factor, see section 3.4.3, chapter three) was not successful.

8.5 Discussion and Conclusions

In summary, the above sections have provided evidence that the Catalogue of Errors does indeed help developers who use it not only to make fewer errors (or to prevent more errors) but also to remove errors sooner in the development rather than later. Given that many studies consider the lack of errors in an artifact to be the principal
determinant of software quality, it can be said that the Catalogue of Errors has a positive impact on software quality (Fenton & Pfleeger, 1997; Kitchenham, 1996; Conte et al., 1986; Diaz & Sligo, 1997; Pfleeger, 1996; Wohlin, 1998). Quality is an important and desirable characteristic of software in general and software development artifacts in particular.

However, as shown in section 8.4, in general, the Catalogue of Errors had a negative impact on the productivity of developers. Specifically, the Catalogue of Errors seems to adversely affect the productivity of developers when producing requirements and code artifacts. The Catalogue was not observed to have an effect on the productivity of developers when constructing design artifacts. Productivity is an important consideration for software development (Boehm, 1987b).

The literature suggests that the production of quality software and the timely delivery of such software or its cost can sometimes conflict with each other. It is argued that as the quality of software development artifacts is increased by producing artifacts with fewer errors in them and by identifying and correcting any injected errors sooner rather than later, the amount of rework on software due to uncorrected errors decreases. Ultimately, the decreased amount of rework translates into rework cost and time savings and a better reputation of developers in the software development industry (Slaughter et al., 1998). This chapter has demonstrated that the incorporation of a Catalogue of Errors in software development can be a powerful tool not only to help lessen the number of errors but also to help the correction of any injected errors sooner rather than later in the development. However, the use of the Catalogue of Errors, did have a drawback. It did affect the productivity of developers when producing software artifacts by slowing them down. Hence, relatively error free software artifacts would take longer to develop. This may not be attractive to those who emphasize the importance of the timely delivery of artifacts by saying that “I'd rather have it wrong than have it late. We can always fix it later.” (Paulk et al., 1994).

While erroneous artifacts can indeed be fixed later, two possible implications are likely to ensue. First, if an erroneous artifact is an intermediary non-executable development artifact (e.g. requirements, design etc.), fixing errors later will definitely lead to an increased cost. Current research is consistent in finding that, as errors are left undetected
and uncorrected, their cost of correction will increase by orders of magnitude (Hevner, 1997). Second, if the erroneous artifact is an executable artifact (e.g., code artifact that is used by a client), the presence of errors is likely to cause disruptions in the operations of the artifact when the client uses it in his/her day to day business operations. As a consequence, while fixing errors later may be convenient to developers, it is not necessarily convenient to the client in terms of time and cost. This cost must be borne by someone, either the client of the artifact or the developers. In addition, fixing errors after the completion of an artifact can also have an adverse effect on the reliability and reputation of the developers. The willingness of a developer to fix errors in an executable artifact after it has been deployed in the client's site may not be sufficient to convince clients to take their business back to that developer. Therefore, while fixing errors after development can have the short-term advantage of impressing clients for timely delivery, it also may have very costly repercussions for the long term.

This discussion suggests that the adverse effect of the Catalogue of Errors on the productivity of developers does eventually pay off, because it attempts to build quality into the software artifact. It is also possible that as developers become more and more proficient with the use of the Catalogue of Errors, they are more likely be more efficient in its use as well. Higher efficiency may improve productivity. This, however, is a future research direction that could be pursued in studies by employing the same group of participant developers using the Catalogue of Errors in more than a single project.
9. Outcomes

9.1 Overview

The objective of this thesis was to develop an error prevention approach using a Catalogue of Errors that encompasses the entire software development and is relatively easy to use, inexpensive and that suits the needs of individual developers. This objective was important because it constitutes an attempt to address a crisis in software development today (Conwell et al., 2000; De Champeaux, 2002; Glass, 2002; Schulmeyer, 1990). The thesis objective was formulated after an extensive literature examination was carried out. This examination demonstrated that, in software development, errors are targeted by using two main approaches, namely, error detection and error prevention approaches. Also, the examination concluded that error prevention can be cheaper and better than error detection, because “A prevented bug [error] is better than a detected and corrected bug [error] because if the bug [error] is prevented, there is no code to correct. ... The thinking that must be done to create a useful test can discover and eliminate bugs before they are coded—indeed, test-design thinking can discover and eliminate bugs at every stage in the creation of software, from conception to specification, to design coding and the rest. (Beizer, 1990, p. 3)”. However, although there is widespread agreement in literature about the above conclusion, in general, error prevention approaches are closely related to and depended upon error detection approaches. Mainly, this dependence consists of knowledge about errors that error detection approaches can generate. This same knowledge is indispensable for error prevention approaches to be successful. While research concerning the development of error prevention was found to have received significant attention during the past decade, the outcomes of this research were generally found to be complex and expensive prevention approaches which required extensive error data collection and analysis and that would be difficult for individual developers to use.

The comparison between error prevention and detection and the problems inherent in the existing error prevention approaches formed the basis of this research, i.e. building an error prevention approach using a Catalogue of Errors.
In order to achieve this objective, three research questions were formulated which complemented each other in the sense that the outcome generated by addressing one question was vital in addressing the following question. Specifically, the first two research questions were conceptual in nature, the first developing a framework which was used by the second research question. Together, they prepared the foundation for the development of a Catalogue of Errors. The third research question constituted the empirical evaluation of the effectiveness of the Catalogue of Errors in helping developers to prevent errors from being injected into their software development artifacts and to remove any injected errors early in the software development. This research question also addressed the impact of the Catalogue of Errors on the productivity of developers. The third research question was addressed by using field experiments.

The objective of this chapter is to present the outcomes of the thesis. In section 9.2, the findings of the research are examined in relation to the initial research questions and objective. The contributions made by addressing each research question are also highlighted. In section 9.3, the limitations of the research are summarised. Future research directions are proposed in section 9.4. Finally, in section 9.5 the thesis is concluded.

9.2 Findings in Relation to Initial Research Questions and Contributions

In chapter three of this thesis, the research questions were posed. This section shows the extent to which this research has achieved what it was originally set out to do and also highlights the main contributions that the research has made.

9.2.1 Research Question One: Findings and Contribution

Research question one states:

What type of error information is important to help developers learn about errors and how can such information be organised into a generic Error Framework?

The objective of this question was to identify the perspectives from which errors can be analysed and integrated into a generic Error Framework. The investigation of this
question is important because it can help understand what type of information is important to be known about errors and the reasons why such information is important. This can help developers to prevent errors or detect them sooner rather than later in the software development. The determination of the error perspectives constituting the Error Framework was informed by the review of literature. It was based on the identification of seven orthogonal perspectives which included error name, origin, severity, cause, prevention guidelines, symptom, and trigger. These were sub-categorised into groups, for example, error name and severity were categorised under identity perspectives; error origin, cause, and prevention guidelines were categorised under causal perspectives; and error symptom and trigger were categorised under diagnostic perspectives. A comparison between the Error Framework developed in this thesis and other existing error frameworks showed not only were previous error frameworks not comprehensive, but they were inconsistent with each other with regard to the analysis of errors.

The usefulness of the perspectives of the Error Framework was then empirically validated with an Error Framework Evaluation Questionnaire. Results suggested that while many participants regarded the error perspectives to be relatively useful, they also indicated the need for additional descriptions of errors in terms of explanations, clarifications, examples, etc. This constituted a valuable feedback given that the Error Framework was to be used as a template to document information about specific errors and to compile such information in a Catalogue of Errors (i.e. outcome of research question two).

The usability of the perspectives of the Error Framework was also assessed. This assessment indicated that, in general, the perspectives were usable, however, the error origin and error severity perspectives were found to be less usable than other error perspectives. This was due to the difficulties that these two perspectives presented when they were used to build up information about errors in the Catalogue of Errors (see chapter seven).

The development of an orthogonal, comprehensive, useful, and generally usable Error Framework was the main contribution made by research question one.
9.2.2 Research Question Two: Findings and Contribution

Research question two states:

*How can the Error Framework (developed in question one) be used to catalogue errors that are commonly injected in various software development artifacts?*

The objective of this question was to use the Error Framework as a template to create a Catalogue of Errors. The development of a Catalogue of Errors would help developers enhance their awareness about the catalogued errors, such as how to identify, avoid, and remove them. It would also generate insights into the quality of the software development processes.

The content and development of the Catalogue of Errors was the subject of chapter seven of this thesis. In this chapter, the learnability of the Catalogue of Errors was addressed. Evidence indicated that participants using the Catalogue require time to learn the Catalogue and training before they could successfully use it. The learning time could be minimised by allocating collective training time, additional error examples and terminology references.

Empirical evidence was obtained from the Catalogue of Errors Evaluation Questionnaires and anecdotal feedback was obtained from open-ended questions that were included in these questionnaires. In general, a significant number of participants provided positive evaluation of the Catalogue of Errors. Most participants found the Catalogue of Errors helpful:

i) To understand the implementation of the various perspectives of the Error Framework in the catalogued errors,

ii) To use and follow,

iii) To identify and correct errors in development artifacts, and

iv) To improve overall software development skills.

However, there were some participants, who regarded the original version of Catalogue of Errors difficult to use and follow. Anecdotal evidence from the open-ended question, also suggested that the Catalogue of Errors needed:

i) additional examples, and
ii) explanation to help with the terminology, and
iii) additional training time.

These concerns were taken into consideration when producing the revised version of the Catalogue of Errors. The partial replication of the original field study with the revised version of the Catalogue of Errors with a different group of participants in a second field study, indicated that the concerns expressed about the first version had been addressed. For example, the Catalogue of Errors Evaluation Questionnaire results were more positively skewed than the evaluation of the original version of the Catalogue of Errors.

This was considered to be significant progress, because the participants who used the two versions of the Catalogue of Errors are comparable. In addition, evidence collected from the participant responses to the open-ended question about the revised version of the Catalogue indicated fewer concerns.

The development of a Catalogue of Errors that was learnable (possibly due to additional examples, time spent and enhanced interactive teaching), useable, useful, and that properly implemented the perspectives of a generic Error Framework and that enhanced the skills of the developers who used it, was a contribution made by addressing research question two. In addition, the progressive refinement of the Catalogue of Errors from its original to a revised version constituted another contribution to this research question.

9.2.3 Research Question Three: Findings and Contribution
Research question three states:

*What is the impact of using of the Catalogue of Errors on software development?*

In chapter three, this question was broken down into two more specific questions, namely,

v) *Does training software developers with the Catalogue of Errors (developed in question two) help reduce the number of errors injected by them into software*
development artifacts? If yes, can the reduction of the injected errors be quantified? "reducing the number of errors" is defined to mean the following:

e) preventing errors from being introduced into an artifact; or

f) identifying and correcting errors injected in an artifact before the construction of subsequent artifacts starts.

vi) What is the effect of the use of the Catalogue of Errors on the productivity of software developers? Can this effect be quantified?

The objective of this question was to assess whether the Catalogue of Errors had any impact on the ability of developers to make fewer of the catalogued errors in their development artifacts and to correct any injected errors sooner rather than later in the software development. In addition, the impact that the Catalogue of Errors might have on the productivity of developers was also assessed.

This research question helped validate the outcomes of the previous two research questions, and also helped change the way that developers approach the construction of error-free software artifacts. In addition, this question conformed to the empirical validation requirement in software engineering (Tichy, 1998).

Research question three was addressed by conducting two field experiments. Both field experiments provided evidence supporting the hypotheses that the participants who are trained to use the Catalogue of Errors did indeed make fewer errors in their development artifacts, and did identify and correct any injected errors sooner rather than later during the software development.

The field experiments, however, showed that the productivity of the participants who used the Catalogue of Errors was adversely affected. However, it was argued that any improvement of the quality of a development artifact (in terms of fewer errors) that resulted from using the Catalogue of Errors would be likely to outweigh any loss in productivity, thereby making software maintenance less costly and easier.

The outcome of research question three is considered an important contribution to knowledge. It extends Yu's (1998) work with a catalogue of C code errors, and it has also shown that catalogues of errors can be developed and effectively used for pre-code
software development artifacts. In addition, research question three has also shown that although an instance of a catalogue of code errors may be specific to a programming language, it can be equally effective in error prevention and early error detection and correction.

9.2.4 An Error Prevention Methodology Using the Catalogue of Errors
Overall, the contributions resulting from the research support the development of a relatively economical error prevention approach using a Catalogue of Errors. The diagram presented in figure 9.1 presents this error prevention methodology.

9.3 Summary of Limitations
The objective of this section is to summarise the limitations of this research. While the limitations have been identified at various points during the discussions in the previous
chapters, in this section, the limitations are summarised. This is important because it helps to view the research findings and contributions in a more realistic context.

9.3.1 Paradigm and Language Dependency of the Catalogue of Errors
As demonstrated in chapter seven, the Catalogue of Errors is divided into three components, namely, the Catalogue of Requirements, Design, and Code Errors. The design component of the Catalogue of Errors is paradigm-dependent. This is because some of the errors that are included in the Catalogue of Design Errors are specific to the object-oriented paradigm (i.e. reuse errors) whereas others are treated as object-oriented errors only (e.g. strong coupling and weak cohesion errors). The literature suggests that strong coupling and weak cohesion errors are also applicable to procedural software (Robertson, 1993). Also the code component of the Catalogue of Errors includes Java errors only. This is a limitation in the scope of the Catalogue of Code Errors because it cannot be used in developing software by using programming languages other than Java.

9.3.2 Limited Number of Types of Errors Included in the Catalogue
The Catalogue of Errors includes a limited number of requirements, design and code errors. This limitation is likely to limit the usefulness of the Catalogue to address all types of errors. The decision about the types of errors to be included in the Catalogue was dictated by time and cost constraints and also by the ability of the participants (i.e. students enrolled in the Internet and Java Programming unit) to assimilate new information. More errors in the Catalogue would probably have overloaded the participant developers as well as affected their performance.

9.3.3 Decision about the Types of Errors Included in the Catalogue
In chapter seven it was indicated that the decision about the types of errors that would be included in the Catalogue was opportunistic in nature. This decision was based on past experience and on the results of a literature survey on common Java errors. In chapter eight it was shown that more than half of the catalogued code errors were never committed by any of the participants. This indicates that the relevance of the code errors component of the Catalogue is limited. Therefore, it is recommended that decisions about the types of errors that should be included in a Catalogue should be based on
formal pilot field studies. This must be carried out in order to maximise the relevance of the Catalogue.

9.3.4 Limitation with the Useability of Error Origin and Error Severity Perspectives of the Error Framework

In chapter seven it was shown that the error origin and error severity perspectives of the Error Framework are of limited useability. These two perspectives were determined arbitrarily and were largely limited to the projects that the participants carried out in the field studies and field experiments of this research.

9.4 Future Research Directions

The objective of this section is to identify future extensions that can be made to the research reported in this thesis.

9.4.1 Further Empirical Evaluation of the Error Framework

The empirical evaluation of the proposed Error Framework was limited to two groups of participants who were also senior computer science and software engineering undergraduate students. In chapter five it was argued that these participants are comparable to junior Java developers who have just joined the industry. Further empirical evaluation of the Error Framework could be undertaken with senior software development professionals. In addition, while the proposed Error Framework was evaluated with the intent of using it as a template to document information about errors, the evaluation of alternative uses (e.g. use the Error Framework to collected historical data about committed and corrected errors, etc.) of the Error Framework could be another area for future investigations.

9.4.2 Further Development of the Catalogue of Errors

As suggested earlier, time and cost considerations limited the number of errors that were included in the Catalogue. Further work can be carried out to enrich the Catalogue of Errors with additional errors (e.g. syntax errors).

Further development of the Catalogue of Errors, also includes the possibility of automation of the Catalogue of Errors. For instance, an automated Catalogue of Errors
Tool could be integrated with other software development artifact tools to automatically identify errors and to present developers with error related information as artifacts are being developed.

Another potential development of the Catalogue of Errors includes the unification of the error-prevention guidelines into a unique prevention framework for errors that are commonly encountered during the development of software artifacts.

9.4.3 Further Empirical Evaluation of the Catalogue of Errors
The existing Catalogue of Errors was evaluated by using Catalogue of Errors Evaluation Questionnaires, which included 5-point scale agreement-disagreement questions and open-ended questions. Additional insight about the reasons leading to the given responses could be obtained by using smaller samples of participants and by conducting in-depth interviews or case studies or alternative interpretivist methodologies which could help shed additional light into the useability aspects etc. of the Catalogue of Errors.

9.4.4 Further Empirical Evaluation of the Impact of the Catalogue of Errors
Given the positive outcome of the field experiments in addressing question three, additional research could be carried out to identify the impact that the size, format, nature of errors, etc. within the Catalogue of Errors can have on participants from different settings. Field experiments with professional software developers could constitute one area of further investigation. Replications of field experiments constitute an important prerequisite in software engineering that can help with the generaliseability of experimental findings. In addition, further investigations could also be carried out to see whether the performance (including productivity, quality of artifacts produced, etc.) of the developers who use the Catalogue is correlated with Catalogue experience.

9.5 Conclusions
In conclusion, the research that has been described in this thesis has achieved its original objectives. This includes developing an Error Framework, using it to create a Catalogue of Errors and evaluating the impact of the Catalogue of Errors in software
development. It was found that the Catalogue of Errors could have a positive impact on the ability of developers to prevent errors from being injected in software development artifacts as well as to detect them early in development. It was also found that the use of the Catalogue of Errors adversely affects the productivity of developers, the cost of which, however, is likely to be offset by the improvement of artifact quality.
10. References


Software Engineering Laboratory, Computer Science Department, Swiss Federal Institute of Technology.


Crothers, R. (2000). Personal Communication: Committee for the Conduct of Ethical Research, Edith Cowan University, Western Australia.


Eickelmann, N. S., & Richardson, D. J. (1996). *What makes one software architecture more testable than another?* Paper presented at the Second International Software Architecture Workshop (ISA’W-2) and International Workshop on Multiple Perspectives in Software Development (Viewpoints ’96) on SIGSOFT ’96 Workshops, San Francisco, California, USA.


The Information Systems Research Arena of the 90's: Challenges, Perceptions, and Alternative Approaches (pp. 155-173). Copenhagen, Denmark.


318


Appendix A: CATALOGUE OF ERRORS

1.1 Descriptive Catalogue of Errors.................................................................332
1.2 Original and Revised Versions of Catalogue of Errors .........................404
1.3 Additional Errors for Future Upgrades of Catalogue of Errors............472
# Descriptive Catalogue of Errors

## Table of Content

1. **OVERVIEW** .................................................................................................................................................. 333
2. **CATALOGUE OF REQUIREMENTS ERRORS** ................................................................................................. 333
   - 2.1 OMISSIONS ................................................................................................................................................ 333
   - 2.2 INCONSISTENCIES, AMBIGUITIES, AND INCORRECT FACTS ................................................................. 337
   - 2.3 COMMON PERSPECTIVES OF REQUIREMENTS ERRORS AND DISCUSSION ........................................ 343
3. **CATALOGUE OF DESIGN ERRORS** .............................................................................................................. 345
   - 3.1 ALGORITHMIC ERRORS .................................................................................................................. 345
   - 3.2 INTERFACE ERRORS ..................................................................................................................... 348
   - 3.3 REUSE ERRORS .................................................................................................................................. 352
   - 3.4 STRONG COUPLING ERRORS ........................................................................................................ 355
   - 3.5 WEAK COHESION ERRORS ............................................................................................................. 362
   - 3.6 DISCUSSION ......................................................................................................................................... 367
4. **CATALOGUE OF CODE ERRORS** .................................................................................................................. 369
   - 4.1 ABUSE OF NAMESPACES .............................................................................................................. 369
   - 4.2 FAILURE TO ESTABLISH CLASS INVARIANTS ............................................................................. 371
   - 4.3 INCREMENT AND DECREMENT OPERATORS .................................................................................. 374
   - 4.4 CONSTRUCTOR INVOKED OVERRIDDEN METHOD ......................................................................... 375
   - 4.5 HONOURING CONTRACTS OF METHODS INHERITED BY DIFFERENT INTERFACES ...................... 377
   - 4.6 OBJECT INITIALISATION ................................................................................................................ 380
   - 4.7 ACCESSING METHODS VERSUS ACCESSING INSTANCE VARIABLES .................................................. 383
   - 4.8 CAST DOWN THE INHERITANCE HIERARCHY ................................................................................... 385
   - 4.9 CONFUSING == WITH equals() ....................................................................................................... 387
   - 4.10 ISSUES WITH INNER CLASSES ................................................................................................... 389
   - 4.11 MEMORY LEAKS .................................................................................................................................. 390
   - 4.12 THREAD DEADLOCKS ................................................................................................................ 394
5. **REFERENCES** ................................................................................................................................................. 397
1. Overview

The objective of this section is to describe the Catalogue of Errors. Each error of the Catalogue is described from the perspectives of the Error Framework (see chapter six). For example, initially, the context of each error is described with an example. This is followed with the description of possible causes, symptoms, triggers, etc.

The Catalogue of Errors has been divided into three parts including the Catalogue of Requirements, Design, and Code Errors. As the part names suggest, each part is comprised of errors that are likely to be injected in a given phase of software development. For example, the Catalogue of Requirements Errors is comprised of errors that are likely to be injected during the requirements phase of software development. The Catalogues of Requirements, Design, and Code Errors are covered in turn in the sections that follow.

2. Catalogue of Requirements Errors

The objective of this section is to describe the four requirements errors, namely, omissions, inconsistencies, ambiguities, and incorrect facts. In sections 2.1 and 2.2 the various perspectives of the error framework (see chapter six) with respect to the requirements errors are discussed. Section 2.3 outlines the common error framework perspectives for the shortlisted requirements errors. Some issues concerning problems with the selected requirements errors are also discussed.

2.1 Omissions

The presence of omission errors in a requirements artifact indicates that a requirement is missing or that it has not been specified. This results in an incomplete requirements artifact. Schneider, Martin, & Tsai (1992) provide an example of an omission (p. 194):

Requirements Specification Text: In the case of two incoming trains, each of which has access to a siding of adequate length, the slower one is to be routed onto the siding while the faster one continues on the main track.
According to Purchase & Winder (1991), if omissions are not prevented or discovered early in software development, the resultant software will not solve the problem it was designed to address (Bell & Thayer, 1976; Purchase & Winder, 1991).

Lamsweerde (2000) maintains that goal representation must be integrated into the development of requirements artifacts in order to establish requirements completeness (Lamsweerde, 2001). A goal is an objective that must be achieved by the system under consideration (Lamsweerde, 2000, 2001; Letier & Lamsweerde, 2002). The rationale behind this integration is that the requirements are complete if "they are sufficient to establish the goals they are refining." (Lamsweerde, 2000, p.7) (Anton & Potts, 1998; Lamsweerde & Letier, 1998). The principal advantage of incorporating goals into requirements development is that goals provide a criterion for the sufficient completeness of a requirements artifact (Lamsweerde, 2001). A second advantage is that goals help identify not only all functional requirements, but also the non-functional ones (Leveson, 1995; Mylopoulos et al., 1999; Nixon, 1993). Another advantage of goal representation is that it greatly facilitates early phases of software development and provides a rationale for the stakeholders of software, supporting the proposed requirements (Lamsweerde, 2001; Lamsweerde & Letier, 1998; Mylopoulos et al., 1999). A fourth advantage is that goal incorporation into requirements specification helps avoid irrelevant requirements (Lamsweerde, 2001). Goals can be identified by looking for keywords like: objective, purpose, intent, concern, in order to, etc. in the user's statements or documentation (Lamsweerde, 2001). Software goal identification is the first step towards avoiding omissions (Mylopoulos et al., 1999).

The second step constitutes the refinement of goals into subgoals. The subgoals are a lower level representation of goals whose accomplishment leads to the achievement of

---

1 Functional requirements lead to the particular functions or services that the system is expected to deliver (Lamsweerde, 2001). Functional requirements are stated formally and are enforced during the implementation (Mylopoulos, Chung, & Yu, 1999).

2 Non-functional requirements are global qualities of the proposed software such as flexibility, maintainability, useability, extensibility, performance, security, safety, etc (Lamsweerde, 2001). Non-functional requirements are normally stated informally and are often contentious. For example, performance goals usually interfere with flexibility goals (Mylopoulos et al., 1999).
the goals from where they are refined (Mylopoulos et al., 1999). Anton & Potts (1998) argue that "Goal refinement is intended to reduce the risk of incomplete requirements." (p. 157). Two types of subgoals are recognised. AND-refinement subgoals relate subgoals to a goal in a way that satisfying all subgoals is necessary for satisfying the goal (Lamsweerde, 2001). OR-refinement subgoals relate subgoals to a goal in a way that satisfying one of the subgoals is sufficient for satisfying the goal (Lamsweerde, 2000). For example, Lamsweerde (2000) defines some requirements of the San Francisco Bay Area Rapid Transit software system (BART) (Lamsweerde, 2000, 2001).

One of the goals of this system, is to serve more passengers. This goal can be satisfied by accomplishing either one of its two subgoals (OR-refinement), namely, trains must be more closely spaced OR new tracks need to be added. Upon completion of goal refinement, goals, subgoals (and, depending on the scope of the software, sub-subgoals, etc.) can be organised into a goal tree or matrix (Lamsweerde, 2001; Pressman, 1997). The organisation of goals, subgoals, etc. into a tree or matrix shows not only how subgoals are derived, but also ensures the traceability of the goals to subgoals.

Goal conflict identification and resolution constitute the third step to prevent omissions. It is possible that conflicts between goals may exist (Easterbrook, 1994). A goal conflict is a situation whereby the accomplishment of one goal rules out the accomplishment of another (Lamsweerde, 2000). For example, in a scheduling software system, the goal of making timetables publicly available competes with security or privacy goals.

The fourth step of preventing omissions, requires the operationalisation of goals into requirements (Mylopoulos, Chung, & Nixon, 1992). Goal operationalisation entails the translation of goals into operational functions and constraints for the final requirements artifact document. In Loy & Mitchell's (1990) ATM example the goal to,

Ensure secure Transactions

translates into the following operationalised constrained requirement:

Requirement: Validate User

Required Precondition: Valid ATM Card is used, Valid PIN is used

Required Postcondition: Valid ATM Card is used, Valid PIN is used

Depending on the scope and the nature of the software being developed, the number of goals and requirements may grow considerably. This may exacerbate the organisation of
goals and requirements and even encourage the introduction of omission errors. It is, therefore, necessary that goals and requirements be organised into traceability matrices. An example of the traceability matrix between goals and requirements is presented in Table 1.

**Table 1 - Goal – Requirements Traceability Matrix**

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Ensure Secure Transactions</th>
<th>Keep detailed transaction information</th>
<th>Etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validate User</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate Transaction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Produce receipt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save transactions</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Information into database</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Traceability is discussed in Landis et al., (1992); Palmer, (1997) and the arguments for developing goal-requirements traceability matrices include: 1) that they provide a justification to stakeholders for specified requirements; 2) that they help in the verification and validation of requirements; 3) that they provide an audit trail; and 4) that they help in detecting requirement and goal conflicts.

The symptom of omission errors is the absence of the specification of required elements in the requirements artifact. McGregor (1998a) sheds light on omission error symptoms and a possible way to expose them. For example,

"A [requirements artifact] model is complete (i.e. lacks omission errors) if no required elements are missing. It is judged by determining if the entities in the model describe the aspects of knowledge being modelled in sufficient detail for the goals of the current portion of the system under development. This judgement is based on the model's ability to represent the required situations and on the knowledge of experts." (p.21).

McGregor's statement implies that in addition to ensuring that complete requirements have been developed for every goal, omission errors can also be triggered by getting the users of software involved in the review of requirements artifacts (Saiedian & Dale,
2000). This is because the users are the knowledge experts as they know what the required software should do.

Porter, Votta, & Basili (1995) suggest that the presence of omission errors can also be exposed by subjecting the requirements artifact to the following types of questions?

i) Are the described requirements sufficient to meet the system goals?

ii) Have all requirements been provided with the necessary inputs?

iii) Have all undesired system states and events been considered, and the appropriate responses specified?

Using questions to trigger omissions is also supported by Potts, Takahashi, & Anton (1994) in their Questions, Answers, Reasons technique and by Lanubile (1998) and Shull (1998).

2.2 Inconsistencies, Ambiguities, and Incorrect Facts

An inconsistency error exists if information within one part of the requirements artifact disagrees or contradicts with other information in the same artifact (McGregor, 1998a). An example to illustrate an inconsistency error in a requirements artifact (RA) could be as follows:

Requirements text in section X of RA: ...an address must include a single unique four digit postcode integer ...

Requirements text in section Y of RA: ...postcode 61512 is located in Western Australia...

Inconsistency Error: The 5 digit postcode integer in section Y of RSA is clearly inconsistent with the requirement in section X that postcodes should consist of four digits only.

An inconsistency error may exist with respect to software behaviour, characteristics, terminology used in the RA, and the timing characteristics of sequences of functions or events (Davis, 1993a).

An ambiguity error constitutes the unclear representation of a concept in a requirements artifact. Davis (1993a) illustrates ambiguity errors in the context of an Air Traffic
Controller (ATC) software system, which uses two aircraft locations display formats. The small display format displays information about the flight carrier, number, altitude, heading, and destination. In order to reduce clutter, the large display format displays only the flight carrier and number information (pp.182-183):

Requirements text: For up to 12 aircraft, small display format shall be used. Otherwise, the large display format shall be used.

Ambiguity Error: The ambiguity exists in “for up to 12”, which some people may interpret as “for up to and including 12” while others as “for up to and excluding 12”.

Such ambiguity errors can cause confusion and misunderstanding and different interpretations about the actual software behaviours and characteristics (Schneider, Martin, & Tsai, 1992).

Incorrect fact errors are errors whereby a requirement asserts a fact that contradicts general domain knowledge or the conditions that have been specified for the software being developed (Lanubile, Shull, & Basili, 1998). An example of an incorrect fact in the requirements specification artifact of a Traffic Light Control software system is as follows:

The user requires that: The traffic light should have the following sequence of colours: Green -> Amber -> Red -> Green

Requirements Specification Text: The traffic light should have the following sequence of colours: Green -> Red -> Green

Incorrect Fact Error: Clearly, the warning Amber has been omitted from the sequence of colours of the traffic light.

If incorrect fact errors are not prevented from being injected into requirements artifacts or uncovered early, developers might end up in producing software with incorrect features or features which the user does not need (Purchase & Winder, 1991).
Lamsweerde (2000) suggests that one way to prevent inconsistency and ambiguity errors from being introduced into requirements artifacts is by using viewpoints and scenarios (Kaindl, 1995; Potts, 1995; Potts, Takahashi, & Anton, 1994). Viewpoints constitute all end-users or other systems that are interfaced to the software system whose requirements are being specified (Kotonya & Sommerville, 1996). For example, in the ATM software system described by Kotonya & Sommerville (1996) some of the identified viewpoints include the bank manager, the home customer, the foreign customer, security officer, etc. Each viewpoint has an associated set of functional and non-functional requirements and constraints. For example, in the ATM system, the bank manager requires transaction reports, a home customer requires cash withdrawals, deposits, balance inquiries, etc. Sometimes, conflicts may exist between different viewpoints (Easterbrook, 1994). For example, in the ATM system the security officer viewpoint requirement that software needs to be maintained regularly interferes with the availability requirement of the home customer and bank manager viewpoints. Kotonya & Sommerville (1996) propose that conflicts need to be identified, compared and prioritised on the basis of importance, and resolved.

A viewpoint may interact with the software system, in which case information between the two may be exchanged (Kotonya & Sommerville, 1996). An exchange of information between a viewpoint and the system constitutes a chronological sequence of events and exceptions, which results in one or more of the viewpoint's requirements to be achieved. This chronological sequence of events is called a scenario. The example in table 2 has been adopted from Rumbaugh, Blaha, Premerlani, Eddy, & Lorensen, (1991) and illustrates two possible scenarios of interaction between the home customer viewpoint and the ATM system. These scenarios both achieve the cash withdrawal requirement of the home customer viewpoint.

---

3 Kotonya & Sommerville (1996) recognise two types of viewpoints, namely direct and indirect viewpoints. Direct viewpoints comprise system end user or other subsystems which are interfaced with the required system. Indirect viewpoints include requirements that constrain the services delivered by the direct viewpoints (Kotonya & Sommerville, 1996).

4 Jacobson et al. (1992) use the term “use case” to represent scenarios and the term “actor” to represent viewpoints. In this context Jacobson et al. (1992) claim that: “Each use case constitutes a complete course of events initiated by an actor and the system. A use case is thus a special sequence of related transactions performed by an actor and the system in dialogue.” (p.159)(similar definition in (Rumbaugh, 1994)).
Table 2 - Example of Scenarios for the ATM software system (Rumbaugh et al., 1991)

<table>
<thead>
<tr>
<th>Normal ATM Use Scenario</th>
<th>ATM Use Scenario with Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ATM asks user to insert card; user inserts cash card</td>
<td>1. ATM requests user to insert card; user inserts cash card</td>
</tr>
<tr>
<td>2. ATM accepts the card and reads its ID number</td>
<td>2. ATM accepts the card and reads its ID number</td>
</tr>
<tr>
<td>3. ATM requests password; user enters '1234'</td>
<td>3. ATM requests password; user enters 9991</td>
</tr>
<tr>
<td>4. ATM verifies the card ID number and password with central computer; central computer ID is details</td>
<td>4. ATM verifies the card ID number and password with central computer; central computer rejects password as incorrect</td>
</tr>
<tr>
<td>5. ATM asks user to select transaction type (deposit, withdrawal, etc.); user selects withdrawal</td>
<td>5. ATM indicates incorrect password and requests user to re-enter it. User enters 1234...</td>
</tr>
<tr>
<td>6. ATM asks for the amount of cash; user enters AUD100.00</td>
<td>6. ATM asks user to re-enter it. User enters 1234...</td>
</tr>
<tr>
<td>7. ATM requests central computer's approval for requested amount</td>
<td>7. ATM requests central computer's approval for requested amount</td>
</tr>
<tr>
<td>8. ATM dispenses cash and requests user to collect it; user collects cash</td>
<td>8. ATM dispenses cash and requests user to collect it; user collects cash</td>
</tr>
<tr>
<td>9. ATM asks whether user wants to continue; user indicates no</td>
<td>9. ATM asks whether user wants to continue; user indicates no</td>
</tr>
<tr>
<td>10. ATM prints receipt, ejects card and requests user to collect them; user collects them</td>
<td>10. ATM prints receipt, ejects card and requests the user to collect them; user collects them</td>
</tr>
<tr>
<td>11. ATM asks user to insert card</td>
<td>11. ATM asks user to insert card</td>
</tr>
</tbody>
</table>

Upon generation, scenarios may be grouped into categories or families of related scenarios (Potts et al., 1994). For example, both scenarios presented in table 2 may be grouped under the category of home customer scenarios.

Potts (1995) suggests that the use of scenarios can be a problem if they are generated in an analytical vacuum and ignore all end-users of the software being developed or other systems interfaced to it. This suggests that each viewpoint may have its own possible scenarios. Therefore, viewpoints and scenarios must be used in conjunction with each other. In order to ensure that all possible scenarios have been produced for all viewpoints and that any produced scenario belongs to a valid viewpoint, a viewpoint scenario traceability matrix may be generated (see table 3).

Table 3 - Viewpoint - Scenario Traceability Matrix

<table>
<thead>
<tr>
<th>Viewpoints</th>
<th>Normal ATM Use Scenario</th>
<th>ATM Use Scenario with exception</th>
<th>Etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank manager</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Home customer</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Foreign customer</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Security Officer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The benefits arising from the use of the Viewpoint Scenario Traceability Matrix are identical to the benefits that were discussed with respect to the Goal-Requirements Traceability Matrix in the section covering omission errors (see section 2.1).

The next step is aimed to help avoid incorrect fact errors. Davis (1993a) suggests that incorrect fact errors are difficult to avoid because “there is no real way of teaching this quality [correctness, i.e. avoiding incorrect fact errors], since it depends totally on the application at hand.” (p. 182). One of the best techniques is to develop a prototype of the required software and to involve the users to act as domain experts (or oracles) (Davis et al., 1993b). A prototype presents the users with a cheap and executable imitation of the required system where they can see directly whether the requirements provided by the prototype correctly fulfill their needs (Mason & Carey, 1983; Saiedian & Dale, 2000). By nature, prototypes are meant to accomplish functional requirements, while cutting corners on non-functional requirements such as reliability, extensibility, algorithmic efficiency, etc. Prototype development has been found to be particularly helpful in avoiding incorrect fact errors in interactive software systems (Mason & Carey, 1983). While there is growing consensus that prototype development is a critical prerequisite to specify correct requirements, it is also notably expensive. This is because, typically a prototype is meant to be a disposable or throw-away sample of the required software (Andriole, 1994; Mason & Carey, 1983; Saiedian & Dale, 2000; Sommerville, 1996). Another danger associated with prototype development is that sometimes developers choose to evolve a ‘good’ prototype into the required software, allowing less-than-ideal solutions to be included (Pressman, 1997). An extensive discussion about prototype advantages and benefits can be found in Carey & Mason, 1983; Urban, (1992).

Potts (1995) suggests that one important advantage of scenarios is that they can illuminate, before coding, how the proposed software would actually affect user goals in different usage situations. Scenarios are also said to be helpful for validation purposes (Fickas, Karat, Johnson, & Potts, 1994; McGregor, 1998a). One problem inherent in the use of scenarios relates to the likelihood of combinatorial explosions when all possible behaviours are enumerated (Lamsweerde, 2000). Lamsweerde (2000) also suggests that the fact that scenarios are procedural in nature may lead to overspecification. As far as viewpoints are concerned, Lamsweerde (2000) suggests that their consideration helps
point out possible inconsistencies in requirements, while Kotonya (1996) criticizes the tendency for viewpoints to be considered in isolation from other viewpoints, detracting from any possible interactions between viewpoints. For example, in a banking software the interaction between a bank customer viewpoint and a bank manager viewpoint, whereby the manager allows the customer to overdraw his account, may not be very clear.

According to McGregor (1998a) one way to trigger inconsistencies is to review the requirements artifact with the objective of ensuring that all concepts, (i.e. goals, requirements, viewpoints, scenarios, and their likely relationships etc.) are consistently described.

Davis (1993a) suggests that one way to trigger the presence of ambiguity errors in a requirements artifact (RA) is to shortlist any statements in the RA which are not clear. If two or more people are then asked to interpret such statements, and more than a single interpretation is obtained, the statements are ambiguous. Still, according to Davis (1993a), ambiguity error can also be identified by ensuring glossaries of terms are included into the RA. In this way, terms in the body of the RA that do not conform with the definitions provided in the attached glossary must be changed or respecified (Bell & Thayer, 1976).

McGregor (1998a) argues that:

"A [requirements artifact] model is correct if it is judged to be equivalent of some reference standard that is assumed to be an infallible source of truth (an oracle in the testing jargon). The standard often is a human expert who judged based on his knowledge. In OO software projects, this oracle is often referred to as a domain expert." (p.21).

Thus, getting the users of the required system involved in the specification of the requirements artifact may also help expose both inconsistency and ambiguity errors (Saiedian & Dale, 2000).
Porter, Votta, & Basili (1995) propose the use of checklist questions to identify the
presence of inconsistency, ambiguity, and incorrect fact errors. Table 4 summarises

<table>
<thead>
<tr>
<th>Inconsistency Errors</th>
<th>Ambiguity Errors</th>
<th>Incorrect Fact Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Are all requirements mutually consistent?</td>
<td>1) Are the individual requirements for all viewpoints stated so that they are discrete and unambiguous?</td>
<td>1) Are all the described functions correct?</td>
</tr>
<tr>
<td>2) Are all requirements consistent with the overall system, and the respective scenarios?</td>
<td>2) Are the all scenarios described clearly and unambiguously?</td>
<td></td>
</tr>
<tr>
<td>3) Are all requirements consistent with the actual operating environment?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The use of questions to trigger inconsistencies and ambiguities is also supported by
Potts, Takahashi, & Anton (1994) in their Questions, Answers, Reasons technique, and

2.3 Common Perspectives of Requirements Errors and Discussion

This section presents some Error Framework perspectives that are common among the
short-listed requirements errors and discusses some issues that developers need to
consider concerning requirements errors in general.

The review of literature about requirement errors suggests that omissions,
inconsistencies, ambiguities, and incorrect facts have similar causes. Therefore, it was
decided to summarise common causes in this section, rather than repeat them in the
earlier sections where the errors are discussed (table 5).
As the previous discussions suggest, goals, scenarios, and viewpoints have been used in combination with each other to prevent requirements errors. This is because goals and viewpoints, on one hand, and scenarios on the other, have complementary characteristics. Goals and viewpoints tend to be abstract and declarative, leaving many of the characteristics and properties of the required software implicit. Scenarios, however, are concrete, procedural, and narrative in nature and have a tendency to make the required software characteristics and properties explicit. It is, therefore, believed that using goals, viewpoints, and scenarios helps avoid requirements errors in general and omissions, inconsistencies, ambiguities, and incorrect facts in particular (Lamsweerde, 2001).

Another issue that requires attention is the fact that sometimes the prevention of certain requirements errors precludes the prevention of others. In this context, Davis et al. (1993b) argue that it is impossible to develop a perfect (i.e., error free) requirements artifact, implying that not all requirements errors can be prevented. According to Davis et al. (1993b) this is because the prevention of some requirements errors might lead to the introduction of others. The following examples support this assertion. If ambiguities are prevented, the level of formality will increase to an extent that it might affect artifact understandability\(^5\) by a non-computer expert. Similarly, if all possible omissions are

---

\(^5\) According to Davis (1993b) lack of understandability is another possible requirements error, in addition to omissions, ambiguities, inconsistencies, and incorrect facts, which are more common.
prevented, the conciseness\(^6\) of the requirements artifact might suffer (Davis et al., 1993b). Davis et al. (1993b) suggest that problems like this may be solved by rating errors in terms of their severity (see chapter six), and by preventing higher severity errors at the expense of lower severity ones.

3. Catalogue of Design Errors

The objective of this section is to describe five design errors with respect to the Error Framework perspectives discussed in chapter six. The design errors include algorithmic errors, interface errors, reuse errors, strong coupling and weak cohesion errors. Each error is covered separately in the sections that follow. The final section discusses some issues concerning strong coupling and weak cohesion errors.

3.1 Algorithmic Errors

In software that is designed using an object-oriented or procedural paradigm, methods or procedures are used to carry out the required operations. Before a method or procedure is completed, however, its algorithm must be designed first. An algorithm consists of a set of well-defined instructions to carry out an operation (McConnell, 1993). An algorithmic error consists of a flaw in the design of an algorithm which should accomplish an operation, given certain input. For example, in the following algorithm, `noHrs` should be multiplied by `hrRate`, and not added to it.

```java
public float computesalary(int noHrs, float hrRate) {
    salary = noHrs + hrRate;  //algorithmic error
    return salary;
}
```

Algorithmic errors may be caused by a variety of phenomena. Firstly, they may result from inadequate understanding of the specification of the method or procedure they are written for (Purchase & Winder, 1991). McConnell (1993) suggests that algorithmic errors may also be caused by thin application domain knowledge and fluctuating or conflicting requirements (McConnell, 1993). Purchase & Winder (1991) add to the list of causes by stating that sometimes developers reuse algorithms that have been written by others. In such cases, it is possible that the incorrect algorithm is reused or that the

\(^6\) According to Davis (1993b) lack of conciseness is another possible requirements error, in addition to omissions, ambiguities, inconsistencies, and incorrect facts, which are more common.
correct algorithm is being reused, but it is not properly understood by the developer (Purchase & Winder, 1991). McConnell (1993) suggests that algorithmic errors may even be due to clerical phenomena (e.g. performing the wrong computation, such as addition instead of multiplication etc.) (McConnell, 1993).

If algorithmic errors are introduced, they may result in an unexpected behaviour or an incorrect result. Therefore, although, the input may be correct, the intended output may not occur because of the algorithmic error (Purchase & Winder, 1991).

In order to avoid algorithm design errors, developers should adopt the appropriate algorithm design process. This algorithm design process is comprised of the following stages:

i) Define the problem which the algorithm is meant to solve (Levitin, 2000). McConnell (1993) argues that this problem definition should provide sufficient detail, including input, output, handling of possible errors, and any information that is not explicit in the user's documentation, etc.

ii) Name the algorithm clearly and unambiguously and after the operation it is meant to carry out (McConnell, 1993).

iii) Define the data that the algorithm needs and decide how to organise this data into the appropriate data structure (e.g. array, vector, list, etc.) (Levitin, 2000) (McConnell, 1993).

iv) Design the algorithm 1. If predefined tested algorithms are available they can be reused or adopted to solve the problem at hand (Levitin, 2000).

v) Check and fine-tune the algorithm to ensure that it solves the problem it was designed for (Levitin, 2000; McConnell, 1993). This should be done with respect to a number of criteria, namely, correctness, clarity, executability, terminability, and efficiency. An algorithm is correct 8 if its logic is correct.

---

1 There are different algorithm design techniques (Levitin, 1999). Levitin (1999) summarises the prominent ones, e.g. brute force, divide-and-conquer, decrease-and-conquer, transform-and-conquer, greedy approach, dynamic programming, backtracking, and branch-and-bound. The divide-and-conquer is the best known general algorithm design technique whose objective was to partition a problem into smaller subproblems. The combined solution of the smaller subproblems constitutes the solution of the original problem (Baase, 1988; Dale, Weens, & McCormick, 2000; Rumbaugh et al., 1991). This algorithm design technique was recommended to the participants of the experiment.

8 Baase (1988) suggests that correctness may be established via formal techniques, which require formal mathematical proofs, and informal techniques.
(Mitchell, 2000). An algorithm is clear if it is simple and it contains precise instructions (Baase, 1988; Levitin, 2000; Rumbaugh et al., 1991). An algorithm is executable if the steps it prescribes can actually be carried out in practice (Dale et al., 2000). An algorithm is terminating, if eventually it terminates its execution (Dale et al., 2000). Finally, an algorithm is efficient if it prescribes the best possible way to solve the problem, in terms of the number of computations and number of variables used (Baase, 1988; Bentley, 1984; Rumbaugh et al., 1991).

vi) Plan how the correctness of the algorithm will be validated (e.g. plan how the algorithm will be tested by producing test cases and expected results) (McConnell, 1993).

vii) Assess the algorithm, bearing in mind the definition of the problem. This assessment should help alert the designer to the presence of injected algorithmic errors. If the result of the assessment indicates that the problem has not been adequately addressed, the process must restart (Levitin, 2000). Robertson (1993) refers to the process of algorithm assessment as desk checking. Desk checking is the process of walking test cases through the algorithm, and keeping a step-by-step record of the content of the various variables involved as the test cases pass through the algorithm logic in order to obtain an actual result(s) for each test case. Discrepancies between actual and expected results suggest the presence of algorithmic errors (McGregor, 1998a; Robertson, 1993).

In addition to avoiding the appropriate algorithm design process, developers can avoid algorithm design errors by establishing pre- and post-conditions for algorithm execution (Meyer, 1997). While pre-conditions assert what should be true before an algorithm starts, post-conditions assert what should be true after the algorithm terminates. Pre-conditions should be ascertained at the beginning of a method or procedure, whereas post-conditions should be ascertained before the method or procedure exits (Payne, Schatz, & Schmid, 1998). An algorithm should guarantee that its post-conditions are met, if its pre-conditions are met (Meyer, 1997). Evidence of the use of pre- and post-conditions can be found in Eyck, Sampath, & Goldstone, (1998).

9 Desk checking is also known as symbolic execution (Coward, 1983).
The following example illustrates how algorithmic errors can be prevented or detected earlier during algorithm design. Assume that class Employee contains a private attribute named salary and a public method named computeSalary() defined as below:

```java
public float computeSalary(int noHrs, float hrRate) {
    salary = noHrs * hrRate; //possibility ONE
    //salary = noHrs + hrRate; //possibility TWO
    return salary;
}
```

Step 1. Establish (as strict as possible) Preconditions and PostConditions

- noHrs, hrRate, salary > 0 (strict preconditions and postconditions)
- Salary = noHrs * hrRate; (very strict postconditions)

Step 2. Desk check against specifications list:

<table>
<thead>
<tr>
<th>Statement</th>
<th>noHrs</th>
<th>hrRate</th>
<th>Salary (expected)</th>
<th>Salary (actual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Pass (possibility ONE) Salary</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>-4 (error detected)</td>
</tr>
<tr>
<td>N pass ... (possibility ONE)</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>-20 (error detected)</td>
</tr>
<tr>
<td>First Pass (possibility TWO) Salary</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4 (error not detected)</td>
</tr>
<tr>
<td>N pass (possibility TWO)</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>12 (error detected)</td>
</tr>
</tbody>
</table>

Levitin (2000) argues that process awareness is important for two reasons. Firstly, it reminds the developers of the obvious steps of problem solving. Secondly, the process emphasises algorithm design as an iterative process requiring constant reworking.

### 3.2 Interface Errors

Interface errors stem from the way objects interact in an application. Object interaction is based on the way an object is structured. The structure of every object includes a provides section, representing the services that the object offers, and a requires section, representing the services that the object will use during its operation (Ammirati, 2006). This was particularly relevant to the work in this study given that the participants of the experiment were students in a university environment.
Gerhardt, & Dye, 1990; McGregor, 1999b). Two objects interact by one object (the client) calling the required services from another object (the server) (Lewis & Loftus, 1998). If each object’s behaviour is assumed to be correct while in isolation, interface errors can occur in two possible cases (Overbeek, 1994):

i) The client object does not call the server’s services in a “legal” manner (i.e. does not provide the correct types of input arguments); and,

ii) The client does not interpret the server’s correct answers (i.e. returned values) properly.

The following example illustrates interface errors. Suppose the requirements of a system reads as follows:

“At the end of the month the payroll for all employees in the company is taken”

Also suppose that this requirement corresponds to the following scenario:

1. Examine employee n
   1.1 Count no of hours employee n has worked
   1.2 Retrieve hourly rate employee n is rewarded with
   1.3 Salary of employee n is the product of 1.1 and 1.2 above
   1.4 Add salary to total
   1.5 Repeat process for all employees in payroll

The above scenario requires the following classes:

Employee class \{computesalary(), Input: int noHrs, float hrRate, Output: float salary\}

Payroll class \{computetotalsalaries(), Input: employee, Output: float total\}

The conditions for interface errors not to occur are as follows:

- The computesalary() method of the server object (Employee) provides the right parameters (int noHrs and float hrRate, two parameters, noHrs first hrRate second) to the method computetotalsalary() of the client object (Payroll).
The value (float) that is returned by the method computeSalary() of the server object (Employee) is stored in the right type of variable (total of type float) in method computeTotalsalaries() of client object (Payroll).

If, however, the above conditions are breached, interface errors may occur as follows:

```java
class Employee { //instances of this class are SERVER OBJECTs
    //with respect to Payroll objects
    float salary;
    public float computeSalary(int noHrs, float hrRate) {
        salary = noHrs*hrRate;
    }
}

class Payroll { //instances of this class are CLIENT OBJECTs
    //with respect to Employee objects
    Employee emp_1; //only one employee declared for simplicity
    int noHrs = 20;
    float hrRate = 2.0;
    float total;

    public float computeTotalsalaries(Employee emp) {
        total = total + emp.computeSalary(hrRate, noHrs);
        //interface breached, wrong order of parameters
        //total = total + emp.computeSalary(noHrs, hrRate); //if total's type is different from method
        //computeSalary()'s return type,
        //interface is breached
    }
}
```

A number of different conditions may cause the introduction of interface errors. Firstly, the developer's failure to understand that individual correct objects do not necessarily mean correct interactions (McGregor, 1999b). Secondly, sometimes developers do not adopt systematic plans to integrate individually correct objects (Jorgensen & Erickson, 1994; Overbeck, 1994). Another cause may be due to the developer's failure to account for likely polymorphic substitutions during object interactions (Alexander, 1999; McDaniel & McGregor, 1994).

The first step to prevent interface errors from occurring is to identify threads of execution in the design artifact of the proposed software (Ammirati et al., 1990). A thread of execution constitutes a scenario of normal or exceptional usage (Ammirati et al., 1990; Binder, 2000; Jorgensen & Erickson, 1994). A thread describes an aspect of the behaviour of the required system, whose enactment is the result of the interaction of

---

11 See definition of a scenario in Inconsistencies and Ambiguities in section 2.2.
some objects of the system. Such an interaction requires some objects to be clients, and others servers, and is underlaid by a sequential chain\(^{12}\) of method executions. The second step requires pairs of client and server objects to be identified for each interaction of the sequence underlying the identified thread (Ammirati et al., 1990; McDaniel & McGregor, 1994). Each client/server pair needs to be analysed in terms of the number, correctness of order and type, and possible values of the parameters involved in the interaction (Bashir & Goel, 1999; McDaniel & McGregor, 1994). The third step is an extension of the second step and requires further analysis of client/server pairs for situations when polymorphic substitutions are involved (Alexander, 1999; McDaniel & McGregor, 1994).

This is particularly important if the client and server belong to inheritance hierarchies (McDaniel & McGregor, 1994). For example, run-time instances of the subclasses of the client may act as polymorphic substitutes for an instance of the client in its interaction with a server object (Alexander, 1999; McDaniel & McGregor, 1994). Similarly, run-time instances of the subclasses of the server may act as polymorphic substitutes for an instance of the server in its interaction with a client object (Alexander, 1999; McDaniel & McGregor, 1994). Depending on the depth of the inheritance hierarchies where the client and server objects are located, polymorphism may compound the number of possible interactions.

When client and server objects interact, a method is invoked. The invoked method may have parameters and may return a result. In this context, polymorphism should also be considered as a compounding factor for the possible number of interactions when a client and server object interact by passing parameters that are instances of other objects, which are henceforth referred to as object parameters. If the object parameters belong to inheritance hierarchies, the polymorphic substitutions by the subclasses or ancestor classes of the object parameters must also be considered (Alexander, 1999; McDaniel & McGregor, 1994). All possible interactions that may occur due to polymorphic substitutions must be identified and appropriately validated (McGregor, 1999b).

---

\(^{12}\) Jorgensen & Erickson (1994) define such sequences of method calls as Method/Message Paths or simply MM Paths. By definition an MM-path is a sequence of method executions that are linked by
If interface errors are introduced in design artifacts, they may be observed in the form of a client object interacting with a server object attempting to use the incorrect or inexistent method. This means that a method may be attempted to be used with the wrong number, type and order of parameters supplied to it.

In order to expose the presence of interface errors, developers should analyse and validate design artifacts by examining client and server objects and ensuring that they interact via the correct method invocations. In addition, developers should also check that the appropriate method is called with the correct type, number and order of parameters. Sometimes, interface errors introduced into a design artifact may escape to the coding phase. In such cases, besides code analysis and validation checks, developers may use tools that perform these checks automatically (e.g. compilers, etc.) (McDaniel & McGregor, 1994; McGregor, 1999b). However, as languages become more dynamic these checks are not always achieved. The example in table 6 which was adopted from McGregor, (1999b) illustrates this point:

<table>
<thead>
<tr>
<th>Table 6 – Interface errors that escape automatic tools (McGregor, 1999b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>string className = &quot;VetoableChangeListener&quot;;</td>
</tr>
<tr>
<td>Class parameterTypes[] = {Class.forName(className)};</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

In table 6, no automatic tool can check the correctness of the className String. If className is misspelled the error may only be flagged at run-time.

3.3 Reuse Errors

When one or more classes that belong to a library or an application are reused in a context, which is different from the one they were originally designed for, reuse errors may occur. Rosson & Carroll (1996) distinguish between two types of reuse, namely, as-is reuse and specialisation or evolutionary reuse.
McGregor (1992) explains that 'as-is' reuse occurs when the to be reused class exists and possesses the required functionality. Instances of such classes may be used directly or they may be incorporated as part of the implementation of other classes. This type of reuse is considered to be safe and successful because of encapsulation and information hiding. Encapsulation and information hiding prevent instances of the reused class from interfering with instances of the other reusing classes. While McGregor (1992) regards this type of reuse safe, Overbeck (1994) is cautious and warns that the appropriate precautionary steps must be taken by testing the relationships between the as-is reused class and the classes in its new environment (Lange & Moher, 1989). Assuming that the reused class is correct\(^\text{13}\), either the relationships between the reused class or any class that can polymorphically substitute it (e.g. classes in the same inheritance hierarchy), or the interacting class or any class that can polymorphically substitute it, must be considered for testing (Lange & Moher, 1989).

McGregor (1992) explains that specialisation or evolutionary reuse occurs when an existing class provides similar, but not the exact kind of functionality required by the proposed software. In this case, a new class can be incrementally developed by inheriting from the existing class. The new class can then locally define new instance variables or methods as necessary. Evolutionary reuse introduces new and often complex dependencies between reused and new code, which defeats the purpose of encapsulation, and thus, provides an alternative for reuse errors to be introduced (Schwartz, 1990). In this case, reuse errors are favoured by the object-oriented feature of inheritance\(^\text{14}\) which allows a subclass in an inheritance hierarchy to directly access and use instance variables and methods defined in its superclass (Jacobson, Christerson, Jonsson, & Overgaard, 1992; Johnson & Poole, 1988; Marick, 1995b). In such cases, the relationship between the new subclass and the reused superclass must be thoroughly tested.

The following example shows how specialisation or evolutionary reuse can cause errors.

\(^{13}\) This assumption is justified because the reused classes are normally obtained from third party libraries which have been appropriately tested and debugged. The reused classes are thus by themselves correct.

\(^{14}\) McGregor & Sykes (1992) recognise two types of inheritance, namely, inheritance for implementation and inheritance for specialisation. Inheritance for implementation connotes that the
class BaseClass {
    int inherited(int x);
    int redefined() // returns a an int range [1..10]
}

class DerivedClass extends BaseClass {
    int redefined() // returns a an int range [0..20]
    int inherited(int x) // inherited from the base class
}

If method inherited(int x) contains the following algorithm:

```
    if(x<0) {
        x = x/redefined();
        return x;
    }
```

In this case, if method inherited(int x) is called with regard to a subclass object
(DerivedClass), it is possible for a division by zero error to occur.

Typically, reuse errors occur when developers fail to understand the possible ways in
which the properties of the superclass class interact with the properties of a subclass
(Dai, 1995; Kim, Clark, & McDermid, 1999c). Dai (1995) also argues that this cause
can be particularly problematic when the source code of the superclass to be reused is
not available and its documentation is not sufficiently detailed. Another possible cause
for reuse errors is when developers possess a poor understanding of dependencies15 that
are introduced due to inheritance (Schwartz, 1990) or when developers possess a poor
understanding of the to be reused classes and the hierarchies that they belong to and
when they are unable to properly use polymorphism (Purchase & Winder, 1991). Lewis,
Henry, & Kafura (1992) recognise that external pressure on developers (e.g. managerial
influence) to adopt reuse can be another cause for reuse errors. Another reason why
reuse errors may be introduced is when developers do not adopt a systematic reuse
process (Rosson & Carroll, 1996).

---

15 Schwartz (1990) recognises two types of dependencies, namely, semantic and behavioural
dependencies. Semantic dependencies have a direct impact on the meaning of the classes and as a result
the subclasses cannot be properly developed without a comprehensive examination of the superclasses.
Behavioural dependencies influence the behaviour of objects and classes resulting from the inherited
behaviour. This required the internal implementation of superclasses or ancestors to be thoroughly studied
before a class is reused.
In order to avoid reuse errors, developers need to identify and evaluate the usage context (Rosson & Carroll, 1996). This requires (1) identifying the class to be reused and learning more about its functionality by trialing it with a driver program in example applications (Rosson & Carroll, 1996); (2) assessing similarities between what is offered by the to be reused class and what is required by the application, and assessing where the to be reused class will be reused; and (3) studying other aspects of the class to be reused and their impact on the required functionality. It is important that the to be reused class is domain independent and sufficiently abstract to fit into the new environment domain (Baldo, Moore, & Rine, 1997; McGregor & Sykes, 1992; Repenning & Perrone, 2000). The next step requires the developer to determine whether the class is to be reused on an ‘as-is’ basis or whether further specialisation is required (Rosson & Carroll, 1996). The final step requires that, once the reused class is incorporated into the new application, immediate analysis and testing and debugging ensues (Lange & Moher, 1989; Rosson & Carroll, 1996). A similar reuse process is briefly outlined in Lewis, Henry, & Kafura, (1991, 1992).

If reuse errors are actually introduced, their symptoms can be varied and will typically range from unexpected and undesired application behaviour to unknown and unpredictable side effects (Purchase & Winder, 1991) (Schwartz, 1990). Reuse errors can be triggered by reviewing design artifacts and ensuring the following:

i) No errors occur when subclass methods interact with superclass instance variables or methods; and

ii) No errors occur when superclass methods are reused in a subclass environment.

Developers are, therefore, better off preventing them than removing them.

3.4 Strong Coupling Errors

Coupling describes the interdependencies between different objects of a software system. Interdependencies between objects exist due to the interaction between client and server objects (Shtern, 2000). Good software design exhibits low or weak coupling between objects as described by simple explicit interrelationships between objects (Briand, Devanbu, & Melo, 1997; Shtern, 2000; Wilkie & Kitchenham, 2000). Strong or excessive coupling increases the complexity of the interdependencies between
objects making the resulting software harder to understand, change, or correct, while favouring error propagation between objects\textsuperscript{16} (L. Briand et al., 1997; Eder, Kappel, & Schrefl, 1994; Wilkie & Kitchenham, 2000).

Chidamber & Kemerer (1994) have suggested that a simple count of the number of objects that any given object interacts with is sufficient to define coupling between objects (Chidamber & Kemerer, 1994). Briand, Devanbu, & Melo (1997), however, argue that finer-grained descriptions capturing the different types of interactions between objects provide more precise guidance to software designers who wish to produce loosely coupled designs (L. Briand et al., 1997). McConnell (1993) rates various finer-grained levels of coupling into acceptable and unacceptable ones with the objective that such knowledge will help software developers to recognise what types of coupling should be used and what types should be avoided. Eder, Kappel, & Schrefl (1994) recognise three different types of coupling, namely, interaction coupling, component coupling, and inheritance coupling.

**Interaction Coupling**

Interaction coupling occurs when methods invoke each other. Such invocations may involve data sharing and may occur across different types of objects. There are five levels of interaction coupling ranging from content to data coupling (McConnell, 1993; Pressman, 1997; Schach, 1999):

i) **Content coupling** is the strongest and worst form of interaction coupling and occurs when a method can directly access the implementation of another (Eder et al., 1994);

ii) **External coupling\textsuperscript{17}** occurs when methods communicate via a structured global shared data space. For example, methods share the public instance variable of a class:

```java
class Employee {
    double currentTax;
    double yearToDateTax;
}
```

\textsuperscript{16} Also known as the “ripple effect” (Pressman, 1997).

\textsuperscript{17} External coupling is a special form of common coupling that may occur in procedural languages, but is not supported in object oriented languages. Common coupling between two or more procedures exists if they access the same global data (McConnell, 1993). Eder, Kappel, & Schrefl (1994) argue that “In general, we do not consider the passing of information between different invocations of methods of the same object in instance variables as external” (p. 8) or even common coupling.
public void calculateTax(Income i) {
    if (i<=6000.0) {
        currentTax = 0;
    } else if ( i< 20000.0 & i>6000.0) {
        currentTax = i*0.17;
    } else if (..) {
    }
}

public void calculateCumulativeTax() {
    yeartodateTax = yeartodateTax + currentTax;
}

//external coupling example

iii) Control coupling occurs when one method passes data to a second method that dictates what the second method should or should not do. The following example illustrates control coupling:

class Tax {
    double totalTax;
    double currentTax;
    public void applyTax(double yearlyIncome) {
        switch (yearlyIncome) {
        case income1:
            totalTax = totalTax + 0.4*(yearlyIncome);
            break;
        case income2:
            totalTax = totalTax + 0.5*(yearlyIncome);
            break;
        default:
            totalTax = totalTax + 0.3*(yearlyIncome);
        }
    }
}

class Employee {
    Tax taxPayable;
    double totalIncomePerYear;
    int incomeBracket;
    public void calculateTax() {
        totalIncomePerYear = calculateTotalIncomePerYear();
        taxPayable.applyTax(double totalIncomePerYear);
    }
    private double calculateTotalIncomePerYear() {
        //code to calculate total income;
    }
}

The weakness of control coupling is that the passing of the control attribute between the methods (totalIncomePerYear) implies that one method (calculateTax()) controls the internal logic of the other (applyTax()). Control coupling is not prohibited by object orientation but it should be avoided, since the change of the implementation of a method may cause hidden changes to the behaviour of the control coupled methods.
For example, if `totalIncomePerYear` instance variable is changed from a primitive data type into the wrapper class `Double`, any change in the implementation of `Employee`, should have its appropriate reflective change made in class `Tax`.

If however a hierarchy of `Income` objects is created each of which has its own specialised `returnTax()` method, then the implementation of the `applyTax()` method could be simplified considerably, but passing to it an `Income` object and calling the `Income` object's `returnTax()` method. Polymorphism and dynamic binding will kick in and perform a run-time determination of actual type of the `Income` object passed in and the appropriate `returnTax()` method to be called.

iv) Stamp coupling\(^{18}\) occurs when two methods communicate via the parameter passing of data structures, however, the called method uses only selected parts of the data structure that it receives. The following example illustrates stamp coupling:

```java
class Employee {
    double sales;
}
class salesStatistics {
    double accumulatedSales;
    public void addSale(Employee e) { // e is an instance of Employee class
    }
}
```

The method `addSale()` should not take an `Employee` object as a parameter, which leads to stamp coupling, but the value of the relevant instance variable, i.e. `sales` of a particular employee, which should lead to data coupling.

v) Data coupling occurs when two methods communicate only by passing relevant parameters. This is the best form of coupling. The following example illustrates data coupling:

\(^{18}\) Also known as data structure coupling (McConnell, 1993).
class Employee {
    double sales;
    public double getSales() {return sales;}
}

class salesStatistics {
    double accumulatedSales;
    public void addSale(double sale) {
        // sale can be obtained by
        // invoking
        // EmployeeObj.getSales()
    }
}

McConnell (1993), Schach (1999), and Eder, Kappel, & Schrefl (1994) divide the various levels of interaction coupling into acceptable and unacceptable categories. Content and control coupling are considered unacceptable, and should, therefore, be avoided. In general, external and stamp coupling are unacceptable, but exceptions are recognised. For example, for two methods that are externally coupled, their coupling is acceptable if the global data that the methods share is read-only (i.e. a set of constants) (McConnell, 1993). For two methods that are stamp coupled, their coupling is acceptable if one method uses all of the elements of the data structure that is passed in as a parameter by the other method (McConnell, 1993). The only universally acceptable form of coupling is data coupling (McConnell, 1993; Pressman, 1997; Schach, 1999).

**Component Coupling**

Component coupling is another type of coupling that is recognised by Eder, Kappel, and Schrefl (1994). While interaction coupling concerns interactions between methods in the same or different classes, component coupling refers to situations whereby an object of one class is used as (Briand, Daly, & Wust, 1999a):

(i) an instance variable within another class,
(ii) as a parameter of a method of another class,
(iii) as local variable of a method of another class, or
(iv) as a parameter of a method which is invoked from within a method of another class.

In the above list of component interactions, (i) and (ii) are also known as specified coupling, because coupling between object or components is clearly specified in the
interface of the class. Whereas, (iii) and (iv) are also known as scattered coupling, because the entire body of the class has to be closely searched for this type of coupling to be identified. The strongest form of component coupling is called hidden coupling. Hidden coupling occurs when the interaction between two instances of two classes is implicit. For example, cascading method invocations, like the method calls in the statement:

```java
getContentPane().add(new Label("MyLabel"));
```

constitute hidden coupling between an instance of JApplet where `getContentPane()` is called from, and the instance of class container, which is returned by `getContentPane()`. Hidden coupling must be changed into at least scattered coupling. For example, the above line of code can be changed into the following:

```java
Container c = this.getContentPane();
Label l = new Label("MyLabel");
c.add(l);
```

The weakest form of interaction coupling exists when there is no direct coupling between the classes (Eder et al., 1994). Although, no direct coupling is referred as the theoretical optimum of component coupling, in practice no direct coupling may not be always possible to achieve, because normally objects from different classes are expected to interact with each other (Jacobson, Booch, & Rumbaugh, 1999; Jacobson et al., 1992). Eder, Kappel, & Schrefl (1994) consider that specified and scattered component coupling are acceptable, whereas hidden coupling, being the worst type of coupling, may be avoided or changed into scattered coupling.

### Inheritance Coupling

Inheritance coupling concerns objects of classes whereby one class is the superclass and the other is the subclass. Eder, Kappel, & Schrefl (1994) recognise four levels of inheritance coupling, ranging from modification to no inheritance coupling at all (Briand et al., 1999a):

1) **Modification coupling** is the strongest and also the worst form of inheritance coupling. Modification coupling occurs when subclasses arbitrarily and
unrestrictedly change or even delete methods inherited from the superclass. Two types of modification coupling are recognised:

a) **Method signature modification coupling**, which occurs when, not only the implementation, but also the signature of the inherited method is changed; and,

b) **Method implementation modification**, which occurs when only the implementation of the method is changed without any restriction.

ii) **Refinement coupling** is similar to modification coupling, however, it is weaker. This is because subclasses can only change inherited methods as restricted by predefined rules. Two types of refinement coupling are recognised, namely:

a) **Method signature refinement coupling** occurs when the subclass refines the signature and the implementation of at least an inherited method, without violating the semantics of that method. There are two rules that prevent the violation of method semantics after the subclass refines it. The covariant rule states that the return types (e.g. class C) of a refined method may be replaced by subclasses of that type (e.g. class C', which is a subclass of C). The contravariant rule augments what the covariant rule says with the additional condition that the types (e.g. class D') of the input parameters of the refined method may be replaced by supertypes (e.g. class D, which is a superclass of D').

b) **Method implementation refinement coupling** occurs if the refined method in the subclass reuses the code of the superclass method it overrides, in addition to any extra refinement code that it may introduce.

iii) **Extension coupling** is the best form of inheritance coupling, which occurs when the subclass adds new methods and attributes without modifying or refining either the signature or the implementation of any of the inherited methods (Eder et al., 1994).

iv) **No inheritance coupling** occurs when there is no inheritance relationship between the classes.

Eder, Kappel, & Schrefl (1994) suggest that the only form of inheritance coupling that is acceptable is extension coupling (apart from no inheritance coupling) and that the other levels of coupling should be avoided. As suggested by McConnell (1993), Mynatt (1990), Robertson, (1993), Pressman (1997) etc., failure to understand and recognise the
acceptable levels of coupling may increase the chances of developers actually introducing them in their designs.

In general, the levels of coupling that have been found to be unacceptable result in software that is difficult to understand, reuse, and modify. Other implications resulting from the presence of unacceptable levels of coupling in a design include, an increased susceptibility to errors of the affected sections of the design and an increased possibility of error propagation to other sections of the design (L. Briand et al., 1997; Mynatt, 1990; Wilkie & Kitchenham, 2000, 2001). Basili, Briand, & Morasca (1998) suggest that unacceptable coupling levels can be detected by using reviews and inspection of design artifacts. If the presence of such levels of coupling is detected in a design artifact, the redesign of the artifact is warranted, and this is always worse than avoiding unacceptable levels of coupling in the first place (L. Briand et al., 1997; Mynatt, 1990; Shtern, 2000).

3.5 Weak Cohesion Errors

Cohesion describes the relatedness or connectivity of the different elements (i.e. instance variables and methods within a class or statements within a method) that a designer includes in the same class (Shtern, 2000). If all elements of the class serve the same goal of the problem domain, the class has a strong cohesion, whereas, if the elements of the class are unrelated, the class has a weak (poor or low) cohesion (Shtern, 2000). Strong cohesion is desirable, whereas weak cohesion is not. Eder, Kappel, & Schrefl (1994) recognise three levels of cohesion, namely, method cohesion, class cohesion, and inheritance cohesion (Briand, Daly, & Wust, 1997).

Method Cohesion

Method cohesion shows how closely related are the statements included into a method. Method cohesion ranges from coincidental to functional cohesion and is based on Myer's (1978) classical definitions of cohesion (McConnell, 1993; Pressman, 1997; Schach, 1999):

1) Coincidental cohesion is the weakest form of cohesion indicating that all the statements that are carried out in the method have nothing in common besides the fact that they reside in the same method.
Logical cohesion methods include statements with similar functionality that are included together not because they are related, but because they happen to perform similar functions. For example, a method called `inputAll()` that inputs customer names, employee time-card information, inventory data, etc. would be an example of a method with logical cohesion.

Temporal cohesion methods include statements that are combined into a method because they are performed at the same time. For example, a `startup()` method might read a configuration file, initialise a scratch file, set up a memory manager, and show an initial screen. The statements that perform such functionality are placed together simply because they are required at the same time.

Procedural cohesion methods include statements that are connected by the same control flow, i.e. carried out in a specified order. For example, if the users of a reporting system like reports to be printed out in a given order, method `printAll()` may include statements that print a revenue report, an expense report, a list of employee phone numbers, and invitations to a party.

Communicational cohesion methods include statements that make use of the same data but are not otherwise related. For example, method `getNameAndChangePhoneNumber(Employee e)` has communicational cohesion if both the name and the phone number are stored in an `Employee` object.

Sequential cohesion methods contain statements that are performed in a specific order and share data from step to step. For example, method `getFileDataAndPerformComputations()` would have sequential cohesion, because it would sequentially carry out three different steps, namely, open file, read file, perform computations on the same file.

Functional cohesion is the strongest form of cohesion where methods possess sequential cohesion and all statements contribute to a single task or objective. Examples of strong cohesion methods include `cas()`, `getCustomerName()`, `deleteFile()`, `computeLoanPayment()`, etc.

Robertson (1993) recognises that although functional cohesion is the best form of cohesion in practice, it is not easy to achieve. In fact, it is nearly impossible to develop methods with functional cohesion. In this context, McConnell (1993) advocates that methods with functional, sequential, communicational, and temporal cohesion are...
acceptable in practice. Methods with procedural, logical and coincidental cohesion are unacceptable (McConnell, 1993).

**Class Cohesion**

Class cohesion addresses the binding of elements within the same class. Here, the elements of a class are comprised of non-inherited methods and instance variables. Five levels of class cohesion are recognised (Eder et al., 1994):

i) Classes are said to have separable cohesion, if they contain methods that access none of the class's instance variables, nor invoke any other method within the class. In this case, the class represents multiple unrelated concepts. This is the weakest level of class cohesion. The following example illustrates separable cohesion:

```java
class Employee {
    String name;
    String address;
    Date birthDate;
    Date hireDate;

    void computeAge() {}
    void computeSalary() {}
    int computeCompanyRevenue(Project p[]) {}
}
```

The method `computeCompanyRevenue()` takes all projects of a company as input parameters and computes the accumulated revenue of that company. It neither accesses any instance variable of `Employee` nor does it invoke any other methods of `Employee`. To improve cohesion the method `computeCompanyRevenue()` should be factored out and included into a different class, e.g. `Company`.

ii) A multifaceted cohesion class represents multiple related concepts. In such a class at least one method references an instance variable or invokes methods of a different, but related concept of the same class. The following example illustrates a class with multifaceted cohesion:

```java
class Reorder {
    Item reorderedItem;
    Company reorderedForm;
    int discount;
    int quantity;
}
public boolean expectedRevenue(){
}

Method expectedRevenue() calculates that expected revenue by subtracting the discount given for a company from the price of an item and multiplying this difference by the quantity of reordered items. The discount given depends on the company. Therefore, the cohesion of the Reorder class can be improved by including discount into class Company:

class Company {
    String companyName;
    double discount;

    double discount() {}
}

iii) A non-delegated cohesion class comprises instance variables that describe only part of the concept that the class is supposed to represent. The following example illustrates a class with non-delegated cohesion:

class Employee {
    String name;
    Date birthDate;
    Project involvedInProject;
    Employee managerOfProject;

    public double computesalary() {}
    boolean managerIncomeHigherThanAverageInProject() {}
}

In class Employee, the instance variables birthDate and involvedInProject depend directly on the instance variable name. However, the instance variable managerOfProject depends directly by the project referred to by involvedInProject and indirectly on the name instance variable. This constitutes non-delegated cohesion. To improve the cohesion of class Employee the instance variables managerOfProject and method managerIncomeHigherThanAverageInProject() should be delegated to class Project as follows:

class Project {
    Employee managerOfProject;
    Date startDate;
    Date expectedEndDate;

    boolean managerIncomeHigherThanAverageInProject() {}
    
}
Concealed cohesion classes include instance variables and methods such that, if regrouped, might form a new distinct and useful class. The following example illustrates a class with concealed cohesion:

```java
class Employee {
    String name;
    String jobProfile;
    int dayOfBirth;
    int monthOfBirth;
    int yearOfBirth;
    int dayOfHire;
    int monthOfHire;
    int yearOfHire;
    ...
}
```

The instance variables describing the various dates may be factored out to a new class `Date` with instance variables `day`, `month` and `year`. The respective instance variables of the class `Employee` are then replaced by two instance variables of type `Date`, namely, `birthDate` and `hireDate`, as follows:

```java
class Employee {
    Date birthDate, hireDate;
    ...
}
```

Model cohesion is recognised as the strongest level of class cohesion and "represents a single, semantically meaningful concept without containing methods which should be delegated to other classes or attributes which can be factored out into separate classes" (Eder, Kappel, & Schrefl, 1994, p.29).

In their discussion, Eder, Kappel, and Schrefl (1994) argue that class designs where cohesion is separable, multifaceted, non-delegated and concealed are unacceptable, and should, therefore, be avoided. These classes result in designs that are difficult to understand and reuse and they should be replaced with classes with model cohesion.
Inheritance Cohesion

Inheritance cohesion portrays the binding of all elements in a class. Here class elements are comprised of all methods and instance variables of a class, i.e. inherited and non-inherited. Sometimes inheritance is used solely to avoid data and code duplication between otherwise conceptually unrelated classes. For example, in an Elevator Simulator example class Elevator and Floor are both shown to be subclasses of a class called Location. This is the weakest kind of inheritance cohesion and should, therefore, be avoided. The strongest form of inheritance cohesion occurs when subclass-superclass relationships are dictated by conceptual relationships, i.e. inheritance is used to define specialised children classes. For example, in a university library system both classes Students and Lecturers are subclasses of class Person.

Pressman (1997) suggests that, although in practice it may not be necessary to determine precise cohesion levels, failure to understand and recognise what is acceptable and unacceptable may cause developers to design non-single-minded classes and methods (McConnell, 1993; Robertson, 1993). Classes where cohesion is unacceptable will not only result in designs which are poorly organised, hard to understand, debug, reuse and modify (McConnell, 1993), but as research shows, such designs are also subject to an increased susceptibility to errors (i.e. error proneness) (Basili, Briand, & Melo, 1995; Basili, Briand, & Morasca, 1998; Briand, Wust, Ikonomovski, & Lounis, 1999b). Basili, Briand, & Morasca (1998) suggest that unacceptable types of cohesion (either method, class, or inheritance) can be detected by reviews and/or inspections of design artifacts. If weak cohesion designs are produced, the best remedy is re-designing classes to make them more cohesive (Sheth, 2000).

3.6 Discussion

The objective of this section is to discuss some issues that are related to a couple of design errors (i.e. strong coupling and weak cohesion) that have been covered above. First, while strong coupling and weak cohesion both need to be avoided in the design of software, sometimes this is not possible. McGregor (1992) illustrates this with an example: Suppose, class A and class B are specialised and that class A depends on class B for a function that A needs and which is provided by B. This dependency suggests that A and B are coupled. The coupling between A and B can be eliminated by duplicating the required part of the functionality of class B in class A. This duplication
has two effects; first, the coupling (dependency) between class A and B is eliminated, because, class A does not have to rely on class B any longer; second, class A is no longer cohesive, because it performs B’s function(s) in addition to its own function(s). This example suggests that the objectives of reducing coupling and increasing cohesion can compete with each other. In such cases it is the designer’s responsibility to balance such competing objectives (McGregor & Sykes, 1992). For example, Shtern (2000) suggests that coupling has a higher priority than cohesion. However, if designers want to promote class reuse, it would seem that cohesion has a higher priority. Again, whether coupling or cohesion take a higher priority depends on the long term software development objectives.

Second, while polymorphism is widely seen as a feature which simplifies design, it does not help accomplish weak coupling. In this context, McGregor (1992) states that:

"In a strongly typed object-oriented language environment, polymorphism allows instances of several classes to be substituted for instances of the class used in the formal definition of a method." (p.127).

This suggests that if a class A is dependent on class B, and class B is part of an inheritance hierarchy H, class A is actually dependent on all of the classes of inheritance hierarchy H, which can polymorphically substitute class B. Therefore, polymorphism can exacerbate coupling.

Third, while inheritance is a powerful concept enhancing reuseability etc., it is also considered to exacerbate weak coupling (L. Briand et al., 1997; Eder et al., 1994; McGregor & Sykes, 1992). Classes that are located lower in the inheritance hierarchy depend on classes that are above them in the same inheritance hierarchy. Thus, "Changes meant to improve efficiency that are made to classes in the hierarchy are propagated to those classes below. Someone making changes that improve the performance of an object for their application could adversely affect the users of classes defined lower in the hierarchy." (McGregor & Sykes, 1992, p. 128). This suggests that inheritance, while important, increases coupling and also the chances of reuse errors.

The objective of this section is to discuss the Java coding errors shortlisted in chapter seven with respect to the perspectives of the error framework (see chapter six). The following subsections cover these errors in turn.

4.1 Abuse of Namespaces

Java uses the notion of namespaces to identify and manage the different program construction elements. A program construction element is defined to represent an identifiable program part that serves a unique purpose. For example, a method is an identifiable part of a Java program and it serves the purpose of providing a given service. A package serves the purpose of grouping related classes together. Other program construction elements include classes, instance variables, local variables, and labels. The definitions of these program construction elements can be found in Deitel & Deitel, (1999a). In this discussion, a program construction element will also be referred to simply as an ‘element’.

Java recognises six different categories of namespaces corresponding to each program construction element, namely, package names, class names, instance variable names, method names, local variables names, and label names (Arnold, Gosling, & Holmes, 2000; Daconia, Monk, Keller, & Bohnenberger, 2000). In order to give programmers greater flexibility, different namespaces allow programmers to use the same name to represent different program construction elements. For example, the same name may be used to represent a particular instance variable and a method and the compiler uses the context in which a name is used and the element scope rules to determine the element that the name represents. Element scope rules control the accessibility of the element to other elements of the program (Deitel & Deitel, 1999a; Winder & Roberts, 2000). For example, in the statement: `c.v=3.14;`, `v` can only represent an instance variable `c`, because a programmer cannot assign a value to a package, class, method, local variable or label (Arnold et al., 2000).

\[ \text{This is allowed but not recommended (Arnold et al., 2000).} \]
The flexibility to assign identical names to different program construction elements may be abused by developers or sometime developers may not even be aware of how namespacing works (Irvine & Offutt, 1995). Both of these situations may cause errors. Namespacing mechanisms can, therefore, sometime be a pitfall in Java and commonly result in hidden instance variable problems (Daconta et al., 2000). The example in table 7 was adopted from Daconta et al., (2000) and illustrates instance variable hiding. The output of the example shows that the local variable answer has hidden the instance variable answer resulting in an undesired outcome.

| public class Poor { |
| public String answer = "Yes!"; //instance variable answer |
| public void wantMoney() { |
| String answer = "No!"; //local variable answer |
| System.out.println("Would you like $1Million> "+answer); |
| public static void main(String args[]) { |
| Poor p = new Poor(); |
| p.wantMoney(); |
| } |
| Output: |
| Would you like $1Million> No! |

Table 7 - Instance Variable Hiding Example (Daconta et al., 2000)

Awareness of the following conventions help avoid any likely problems that may be associated with instance variable hiding (Arnold et al., 2000; Daconta et al., 2000):

i) Class instance variables may be hidden by local variables of the same name;

ii) Class instance variables may be hidden by subclass instance variables of the same name;

Incorrect values of instance variables may result from instance variable hiding. If abuse of name spaces errors are committed, then developers can trigger the presence of such errors by identifying all variables (including instance variables and local variables) sharing the same name whose scopes overlap. The actual values of the affected instance values are also referred by the name of field and the problem is known as hidden field problem (Arnold et al., 2000).

A local variable includes variables that are declared locally in a method and parameters that are passed in as arguments to the method (Deitel & Deitel, 1999a).

This also includes situations where instance variables are 'multiply inherited' from two or more interfaces or from a superclass and one or more interfaces (Daconta et al., 2000).

---

20 Instance variable are also referred by the name of field and the problem is known as hidden field problem (Arnold et al., 2000).
21 A local variable includes variables that are declared locally in a method and parameters that are passed in as arguments to the method (Deitel & Deitel, 1999a).
22 This also includes situations where instance variables are 'multiply inherited' from two or more interfaces or from a superclass and one or more interfaces (Daconta et al., 2000).
variables can be exposed by using utility methods, such as
\texttt{get\_instance\_variable\_name()} and compared to expected instance variable values.

These errors may be prevented either by changing the names of the concerned variables or by qualifying the instance variable that is hidden with the \texttt{this} or \texttt{super} keywords. In order to avoid hiding an instance variable by a local variable, the instance variable should be qualified with the \texttt{this} keyword (case i), above). For example, in the class definition of table 7, method \texttt{public void want\_money()}, could be modified as follows:

\begin{verbatim}
System.out.println("Would you like \$1Million > "+this.answer);
\end{verbatim}

In this case the instance variable \texttt{answer} rather than the local variable \texttt{answer} will be accessed. In order to avoid hiding a superclass instance variable by a subclass instance variable, the superclass instance variable should be accessed via the \texttt{super} keyword (case ii) above) (Daconta et al., 2000). For example, if class \texttt{Poor} were to have a superclass, say \texttt{Person}, with an instance variable \texttt{answer}, and method \texttt{public void want\_money()} of class \texttt{Poor} needed to access the superclass' \texttt{answer}, then method \texttt{public void want\_money()} would have to be modified as follows:

\begin{verbatim}
System.out.println("Would you like \$1Million > "+super.answer);
\end{verbatim}

In this case, the superclass (i.e. \texttt{Person}) instance variable \texttt{answer} will be accessed.

\subsection*{4.2 Failure to Establish Class Invariants}

Meyer (1997) defines a class invariant as an assertion expressing conditions on the instance variables of a class which must be preserved during the lifetime of every object of that class (Ruby & Leavens, 2000; Webber, 2001). For example, in a \texttt{BankAccount} class which reports an account balance and keeps track of deposits and withdrawals made on \texttt{BankAccount} objects via instance variables \texttt{deposit\_list} and \texttt{withdrawal\_list}, the invariant could be established via the following assertion (Meyer, 1997):

\begin{verbatim}
balance = total deposits\_list transactions - total withdrawals\_list transactions.
\end{verbatim}
Failure to establish and preserve the class invariant signals anomalous class implementation (Andersen, 1996; Hubbard, 1999). The establishment and preservation of the class invariant not only apply to methods that exist in the class, but also to methods that might be added later, thereby allowing control of the future evolution of the class (Meyer, 1997). Stroustrup (2000) suggests that the class invariant is devised by the programmer to reflect the constraints that the requirements impose on the instance variables of the class. This implies that failure to understand such constraints and provide for them in the implementation of a class will cause class invariant errors.

Invariants need to be evaluated at the time of the creation of an object and at the entry and exit points of all methods in a class and must hold whenever methods get called (Lencevicius, Holzle, & Singh, 1997; Paryavi & Hankley, 1995; Payne et al., 1998; Stroustrup, 2000). Class invariants, however, do not need to be enforced during the evaluation of private methods because the private methods eventually get called by public methods and the invariants do not have to hold true during the execution of public methods (Meyer, 1997; Paryavi & Hankley, 1995). In the case of inheritance hierarchies, subclasses may inherit their parent’s invariants; they may also override them or create new ones to suit their specifications (Paryavi & Hankley, 1995). If a class invariant is violated, the program should stop with an error message or throw an exception (Paryavi & Hankley, 1995). The example in Table 8 illustrates the establishment of the invariant of class Date (method invariantO) and one possible way invariants can be evaluated (e.g. method nextDay()).

---

23 Meyer (1997) suggests that invariants for all objects of a class need to be satisfied at all 'stable times'. A stable time is defined as a time when the state of an instance of a class is observable. An observable state exists after the object’s constructor has been executed and after every method call (Meyer, 1997).
As the example in table 8 shows, invariant violation results in incorrect instance variable values which consequently result in invalid or inconsistent object states (Hubbard, 1999). This may ultimately bring about incorrect program output or behaviour (Andersen, 1996; Payne et al., 1998). This implies that in order to trigger errors that violate class invariants, a developer should attempt to exercise suspicious code with values that attempt to violate the constraint imposed by the requirements. In the example shown in table 8, the result (false) generated after the invocation of method invariant() after statement day++; suggests that there is a flaw in the implementation of method nextDay(). This is because the invariant of object d, has been violated as observed by its invalid state (Day: 32, Hour: 16).

---

24 For simplicity this example has illustrated the class invariant being implemented in a method (Invariant()). An alternative to this method consists of implementing the class invariant in a class (e.g. Assert). A fully developed example for this method is presented in Payne et al., (1998).
4.3 Increment and Decrement Operators

Java applies increment (++) and decrement (--) operators to numeric variables or numeric array elements (Arnold et al., 2000). These operators are otherwise called unary operators because they are associated with a single operand (Deitel & Deitel, 2002). The increment or decrement operators increment or decrement the value of the operand by 1, hence the name. If an increment/decrement operator is located before the variable, it is called pre-increment/decrement operator, respectively. If an increment/decrement operator is located after the variable it is called post-increment/decrement operator (Deitel & Deitel, 2002). Table 9 which has been adopted from Deitel & Deitel (2002) summarises the differences between pre-, post-increment/decrement operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Preincrement</td>
<td>++x</td>
<td>After variable x is incremented by 1, the new value of x is used in the expression where x is located.</td>
</tr>
<tr>
<td>++</td>
<td>Postincrement</td>
<td>x++</td>
<td>The existing value of x is used in the expression where x is located first, then x is incremented by 1.</td>
</tr>
<tr>
<td>--</td>
<td>Predecrement</td>
<td>--y</td>
<td>After variable y is decremented by 1, the new value of y is used in the expression where y is located.</td>
</tr>
<tr>
<td>--</td>
<td>Postdecrement</td>
<td>y--</td>
<td>The existing value of y is used in the expression where y is located first, then y is decremented by 1.</td>
</tr>
</tbody>
</table>

In general, the expression x++ is equivalent to x = x + 1. However, there is a difference. In the former case, x is evaluated only once, whereas in the latter, x is evaluated twice. This is not always explained clearly in Java references. Arnold, Gosling, & Holmes (2000) suggest that failure to recognise this fact may cause erroneous implementations. Such erroneous implementations could consist of incorrect values for local, instance variables, etc. The example in table 10 has been adopted from Arnold et al., (2000) to illustrate a possible erroneous situation.
In order to expose possible errors associated with increment and decrement operators, developers should compare the results obtained from statements like `expression++` with the results obtained from statements like `expression = expression + 1`.

Errors associated with increment and decrement operators can occur if the rules associated with increment and decrement operators are violated or ignored. Therefore, they can be avoided by establishing awareness about such rules.

### 4.4 Constructor Invoked Overridden Method

In Java a constructor is a special kind of method that is used to initialise objects of a given class. It is possible that a constructor might call methods of its own to carry out object initialisation (Deitel & Deitel, 2002). For example, the constructor `public C0 {*some implementation*/ .. m();)` of class `C` may call method `public void m0 {*m implementation*/}` of class `C` to carry out a given initialisation task. This practice is commonly used and causes no problems (Deitel & Deitel, 2002; Haggar, 2000). However, non-intuitive incorrect results may be produced, if a new subclass of class `C`, say class `C1`, overrides the method `public void m0 {*subclass implementation*/}` that is being called from the `C1` superclass (i.e. class `C`) constructor (Haggar, 2000). The example in table 11 has been adopted from Gosling (1996) to illustrate this situation.
Table 11 - Example Constructor Invoked Overridden method (Gosling, Joy, & Steele, 1996)

```java
class super {
    super() { printThree();
    }
    void printThree() {System.out.println("Three");
    }
}
//definition of superclass

class Test extends super {
    int indiana = (int)Math.PI;
    void printThree() {System.out.println(indiana);
    }
    public static void main(String[] args) {
        Test t = new Test();
        t.printThree();
    }
} //main method
//definition of subclass

Output will be:
0
3
instead of the expected:
Three
3
```

The table 11 example features superclass `super` with constructor `super()` which calls method `printThree()`. Method `printThree()` is defined in class `super` and prints the value of string "Three". Class `Test` inherits from class `Super`, but it overrides method `printThree()` to print the truncated value of PI (i.e. integer 3), which is stored in an instance variable called `indiana`. Method `printThree()` is called with reference to instance `t` of subclass `Test` and the result of its invocation is displayed. The results are not what one would normally expect.

The unexpected results occur because, during the creation of the instance `t` of class `Test`, the `Test`'s superclass constructor (i.e. `super()`) is implicitly called first (Arnold et al., 2000; Deitel & Deitel, 2002). Constructor `super()`, however, invokes the overriding rather than the original version of `printThree()` in class `Super`. However, when the overriding version of `printThree()` is called, the instance variable `indiana` has not been initialised yet. This results in the `indiana` default value (0) being retrieved by the overriding `printThree()` (Deitel & Deitel, 2002). Haggard (2000) suggests that this type of error might not be very common, however, awareness of it can save considerable time if it is encountered. This phenomenon is also known as a downcall,
because “the superclass “calls down to” the overridden method” (Ruby & Leavens, 2000, p. 208).

Errors associated with downcalls may exhibit non-termination or unexpected or incorrect behaviour, because when the superclass constructor calls down to an overridden method, the overriding subclass method may behave in an unexpected way (Ruby & Leavens, 2000).

According to Ruby & Leavens (2000) one possible cause for constructor invoked overridden methods is associated with developer’s failure to understand issues related to down calls and this may be exacerbated when libraries of a class developed by a third party are extended and their methods overridden.

4.5 Honouring Contracts of Methods Inherited by Different Interfaces

Java allows a class to implement more than one interface (Arnold et al., 2000). If two or more interfaces declare a constant with the same name or specify a method with the same signature, which are then implemented by the same class, problems may occur in the implementing class (Haggar, 2000). These problems are related to the fact that there is only one such constant or method whose implementation will ultimately have to be determined in the class which implements the interfaces. Haggar (2000) defines this situation as an interface name clash problem.

In this context, Arnold, Golsing, & Holmes (2000) state that, “The real issue is whether a single implementation of the method can honour all the contracts implied by that method being part of different interfaces.” (p. 113). The example portrayed in table 12 has been adopted from Arnold et al., (2000) to illustrate the interface name class problem.
In the example in Table 12, it may be hard to implement method `draw()` in class `GraphicalCardDealer` that honours both contracts suggested by the `CardDealer` and `GraphicalComponent` interfaces (i.e. flipping the card AND rendering on the default device) simultaneously. If a Java developer were to implement method `draw()` in class `GraphicalCardDealer` to honour both contracts, then this method would exhibit weak cohesion. The `draw()` method, would therefore, be subject to drawbacks associated with weak cohesion, as discussed earlier in section 3.5.

Errors associated with method clashes in interfaces can be exposed by review or inspection, which can help in identifying the methods that are inherited by more than one interface and that share the same signature. Sometimes compilers can detect these errors. If such methods share the same signature, but not the return type, a compiler error will be the symptom of the clash in method names. For example, if interface `CardDealer` were to be specified exactly as shown in Table 12, and if interface `GraphicalComponent` were to contain method `int draw()` (instead of the original `void draw()`), and the `GraphicalCardDealer` class were to attempt the implementation of `void draw()`, a compiler error will be generated. Name clash problems are normally caused when Java developers attempt to reuse classes from disjunct libraries to construct software (Haggar, 2000).
Haggar (2000) discusses the following possible options to avoid this problem. Firstly, Haggar (2000) suggests that identical name constants declared in different interfaces can be accessed in a class which implements such interfaces, by fully qualifying their names. For example, as shown in table 12, class GraphicalCardDealer can access interface variables as follows:

```java
int var1 = CardDealer.variable1;
int var2 = GraphicalComponent.variable1;
```

Secondly, Haggar (2000) suggests that, if identical signature methods are involved in two (or more) interfaces, developers need to write a new interface which extends existing the ones (i.e. subinterfaces). In the subinterfaces, the names of the method(s) that are likely to clash should be changed. For example, the name class problem shown in the example in table 12 can be resolved as shown in table 13.

| interface MyCardDealer extends CardDealer {
|     void drawASinFlipTopCard();
| } class GraphicalCardDealer implements MyCardDealer, GraphicalComponent {
|     void drawASinFlipTopCard(); //flip top card
|     void draw(); //render on default device
|
|
| Clearly, interface MyCardDealer has been created, which inherits from CardDealer. The MyCardDealer interface declares a new method, namely, `void drawASinFlipTopCard()`, whose objective is identical to CardDealer's draw() method. However, here the name clash problem has been resolved, because, class GraphicalCardDealer can implement MyCardDealer and GraphicalComponent without any method name clash occurring.

This approach, however, can be problematic, because in Java, interfaces can be used as references to objects. If a CardDealer reference is used, it can only have access to the original CardDealer's draw() and not to the MyCardDealer's drawASinFlipTopCard().

---

25 This occurs because: any object which is referenced through the CardDealer interface can only call the methods of this interface or of any other interfaces it extends from (Haggar, 2000).
Thirdly, Haggar (2000) proposes a solution that forgoes a single class implementing both interfaces, in favour of two separated classes implementing both interfaces as shown in Table 14.

Table 14 – Another possible solution to resolve conflicts of interface method names

```java
class MyCardDealer implements CardDealer {
    void draw() {/*code to flip top card*/}
}
class MyGraphicComponent implements GraphicComponent {
    void draw() {/*code to render on default device*/}
}
```

This solution, however, has a direct implication on the coupling and the cohesion of the new classes.

4.6 Object Initialisation

Java has built in mechanisms that help ensure the proper initialisation of the instance variables of a newly created object (Venners, 1998b). Such mechanisms ensure that if instance variables are not explicitly initialised by the developers, Java will automatically award them predictable default initial values which depend on the type of instance variable (Arnold et al., 2000; Deitel & Deitel, 2002; Venners, 1998b). For example, if an instance of type `MyObject` has an instance variable `x` of type `int` which is not initialised by the developer, value 0 will automatically be assigned to instance variable `x`. While this automatic instance variable initialisation reduces the likelihood of initialisation-related errors, it does not totally eliminate them (Bek, 2001; Venners, 1998a, 1998b). Venners (1998a) argues that one cannot rely on default initialisation, rather one should attempt to explicitly and properly initialise all instance variables of an object during its construction. The following example (table 15) was adapted from Deitel, Deitel, & Nieto (2000) and shows that reliance on an improperly initialised object may cause errors.
As illustrated above, the state of the `testDate` object is initialised to an erroneous date and any calls made to the `nextDay()` method keep producing incorrect results.

The example in table 15 also suggests that if objects are not properly initialised, it is possible for them to start their life with an incorrect state, which results from incorrect instance variable values (Venners, 1998a). As objects with incorrect or invalid states are used, they are likely to yield incorrect or unpredictable behaviour (Deitel & Deitel, 2002). The invalid object states can be exposed by allowing developers to bypass Java's encapsulation mechanism using `getInstancevariable()` and `setInstancevariable()` to report and modify the values of the instance variables, respectively.

Venners (1998a) proposes a set of guidelines to help avoid possible problems associated with object initialisation which cause object initialisation errors. The set of guidelines is explained below:

---

16 The programmers of languages like C, C++, etc. must remember to initialise instance variables of objects before they are used. Failure to initialise newly created objects can lead to initialisation errors in such languages (Venners, 1998b).
i) An object's instance variables should be declared private in order to avoid direct access from other objects (Venners, 1998a). Accessor and mutator methods (e.g. `getInstanceVariable()` and `setInstanceVariable()`) should be the only way that the instance variables of an object can be accessed and modified by other objects (Deitel & Deitel, 2002; McGregor & Sykes, 1992). For example, class `MyDate` shown in Table 15 could be equipped with methods such as `void setMonth(int month)` or `int getMonth()` to control access to the instance variable `month`.

ii) The failure of developers to fully understand how Java's mechanism for object initialisation works could be another cause of object initialisation errors (Venners, 1998a). This would require developers to raise their awareness of all initialisation mechanisms (Arnold et al., 2000; Eck, 2001).

iii) Sometimes the cause for incorrect object initialisation relates to the fact that the developers have failed to understand the requirements of an object. For example, a developer coding a `TrafficLight` class may have failed to understand that any object of type `TrafficLight` needs to have three instance variables initialised to three constants representing the three colours of a traffic light: green, amber, red. Venners (1998a) refers to such values as the "default natural values" of an object. While admitting that not every object can have default natural values, Venners (1998b) postulates the guideline that Java developers should initialise object instance variables to natural default values if such values exist. Otherwise, they should use constructors to initialise instance variables to known valid initial values (Irvine & Offutt, 1995; Venners, 1998a).

iv) This is a corollary guideline to guideline number iii). Constructors should be equipped with code which checks the validity of the data used to initialise object instance variables. Such code should be compliant with the specification and exceptions should be automatically thrown when invalid data are detected (Venners, 1998a).

Venners (1998a) argues that the above guidelines are not theoretical; their objective is to help developers to "acquire a mindset conducive to good design" (p. 15) to avoid errors related to object initialisation.

---

27 Also known as get and set methods.
4.7 Accessing Methods versus Accessing Instance Variables

Arnold, Gosling, & Holmes (2000) point out that in Java the type of reference and the type of object the reference is referring to, play an important role in determining the method or instance variable that can be accessed. In general, a Java object does not necessarily have to be referred to by a reference of the same type as the object. For example, inheritance allows a superclass type of reference to refer to a subclass type of object (Deitel & Deitel, 2002). If a method is invoked via an object reference, the actual type (i.e. the class that the object is instantiated from) of the object determines the implementation of the method to be used (Winter & Roberts, 2000). This rule is intuitive.

When instance variables are involved, however, the rule is different and counter intuitive. Arnold, Gosling, & Holmes (2000) state that when an instance variable is accessed, "the declared type of the reference" (p. 75) is used, rather than the actual type of object that the reference is referring to, as one would expect. The example shown in table 16 has been adapted from Arnold, Gosling, & Holmes (2000) to illustrate the difference between the rules of accessing methods and accessing instance variables.
### Table 16 - Accessing methods vs Accessing Instance variables

<table>
<thead>
<tr>
<th>Class Supershow</th>
</tr>
</thead>
<tbody>
<tr>
<td>public String str = &quot;SuperStr&quot;;</td>
</tr>
<tr>
<td>public void show() { System.out.println(&quot;super.show: &quot; + str); }</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Class Extendshow extends Supershow</th>
</tr>
</thead>
<tbody>
<tr>
<td>public String str = &quot;ExtendStr&quot;;</td>
</tr>
<tr>
<td>public void show() { System.out.println(&quot;extends.show: &quot; + str); }</td>
</tr>
</tbody>
</table>

```java
public static void main(String args[]) {
    Extendshow ext = new Extendshow();
    Supershow sup = ext;
    sup.show();
    ext.show();
    System.out.println("sup.str = " + sup.str);
    System.out.println("ext.str = " + ext.str);
}
```

Output:

```
Extend.show: ExtendStr
Extend.show: ExtendStr
sup.str = SuperStr
ext.str = ExtendStr
```

The example shown in table 16 features two classes related by inheritance. The superclass is called Supershow and contains a string instance variable (i.e. str), whose content (i.e. SuperStr) is accessed and displayed by method show(). The subclass is called Extendshow which overrides method show() to access and display the content of instance variable str. The str instance variable has been redefined in Extendshow to a new string value, namely, "ExtendStr".

The example in table 16 also shows the definition of the sup superclass (i.e. Supershow) reference and ext subclass (i.e. Extendshow) reference. The sup reference is pointing to an Extendshow type object. The exact method to be called is determined at run-time resulting in the invocation of the Extendshow version of show(). When sup, which is referencing an Extendshow type object, accesses the str instance variable, the superclass SuperStr rather than the Extendshow ExtendStr instance variable is accessed.

24 In object-orientation terms, this is known as dynamic binding. Dynamic binding is a feature of object oriented software whereby the correct type of an object is determined at run-time, rather than compile-time while automatically determining the correct method that needs to be called (Schoch, 1999).
A lack of awareness of the rules discussed above may cause errors to be introduced in situations similar to the example shown in table 16. These errors exhibit themselves when the wrong value is assigned to the wrong instance variable resulting in subclass and/or subclass objects with possible incorrect or invalid object states.

Establishing awareness of these rules, (especially in the case when instance variable access via superclass and subclass references is concerned) is the first step towards avoiding errors. Another way to avoid errors associated with the above access rules is to always avoid direct access to instance variables. For instance, in the example in table 16 the instance variable is accessed directly:

```java
... public void show() {System.out.println("Super.show: " + str);} ...
```

Instance variable access should be carried out by using specialised methods, such as accessor and mutator methods (e.g. `getInstancevariable()` and `setInstancevariable()`).

4.8 Cast Down the Inheritance Hierarchy

Java allows one type of object reference to be assigned another type of reference. Such assignments, however, are governed by certain rules. When the assignment rules are violated, assignment-related errors occur. An assignment between different types of references is possible if such references refer to objects which have been instantiated from classes that belong to the same inheritance hierarchy. Roberts & Heller (1997) refer to the process of assigning the reference of one type to the reference of another type as *type changing*. Type changing may be performed automatically by the compiler in which case it is called *implicit conversion*29 (Brogden, 1999; Roberts & Heller, 1997). Roberts & Heller (1997) generalise implicit conversion rules by stating that “in general, object reference conversion is permitted when the direction of the conversion is “up” the inheritance hierarchy;” (p.109). This means that in Java it is acceptable to assign a subclass reference to a superclass reference (i.e converting or changing the type

---

29 Arnold, Gosling, & Holmes (2000) refer to implicit conversion as *upcasting*. 
of a subclass reference to a superclass) (Deitel & Deitel, 2002). The code shown in table 17 has been adapted from Roberts & Heller, (1997) to illustrate implicit conversion.

Table 17 – Implicit Conversion (Roberts & Heller, 1997)

```java
class Citrus {
    /* code to implement class Citrus */
}
class Tangelo extends Citrus {
    /* code to implement class Tangelo */
}
class Test {
    Tangelo tangelo = new Tangelo();
    Citrus citrus = tangelo; // implicit conversion
    Citrus citrus2 = new Citrus();
    Tangelo tangelo2 = citrus2; // Error
```

The code in table 17 shows a `Tangelo` being converted into a `Citrus` (`Citrus citrus = tangelo;`). The system allows such conversion because its direction is “up” the inheritance hierarchy. Essentially, this is allowed because of the “is a” relationship between `Tangelo` and `Citrus` established by inheritance in table 17 (Deitel & Deitel, 2002).

However, if conversion “down” the inheritance hierarchy is attempted (e.g. in table 17: `Tangelo tangelo2 = citrus2;`) the compiler will reject it and flag an error. This is because now the relationship `Citrus` is `Tangelo`, implied by the conversion, `Tangelo tangelo2 = citrus2;`, is untrue. In this case, a Java developer can explicitly force the conversion to take place. Conversion or the type changing that is explicitly performed by the Java developer for an assignment that would otherwise be rejected by the Java compiler is called casting\(^\text{10}\) and can be accomplished using the cast operator (Brogden, 1999; Deitel & Deitel, 2002; Roberts & Heller, 1997). In the example presented in table 17, the line which generated the error can be modified to:

```
Tangelo tangelo2 = (Tangelo) citrus2;
```

\(^{10}\) Arnold, Gosling, & Holmes (2000) refer to casting as downcasting.
The newly inserted code (Tangelo) is called a cast operator and its inclusion means that the Java developer is aware that the conversion is invalid for the compiler, but he/she still wishes to go ahead with it.

While casting is common in practice, it can be dangerous (Deitel & Deitel, 2002) (Irvine & Offutt, 1995). For example, the following statement, where the assignment is carried out, survives compilation:

```java
Tangelo tangelo2 = (Tangelo) citrus2;
```

Nevertheless, an error (java.lang.ClassCastException) will be generated and flagged where the statement is located when attempts are made to run the application. This occurs because the `tangelo2` reference is still referring to a `Citrus` type object (i.e. the same object that the `citrus2` reference is referring to (`Citrus citrus2 = new Citrus();`)) prior to the assignment. Deitel & Deitel (2002) and Friesen (2000) imply that cast down inheritance hierarchy errors may occur because Java developers do not fully understand conversion and casting rules or neglect to apply them properly.

Deitel & Deitel (2002) and Haggar (2000) propose that in order to avoid or expose problems associated with casting, the exact type of the object which a reference is pointing to should be verified before the object is allowed to get involved in any operation. Object verification can be accomplished using the Java `instanceof` operator. For instance, checks like the following can be added to code:

```java
if (tangelo2 instanceof Tangelo) {
    /* what to do if object is of type Tangelo */
}
```

Such checks will improve Java developers' chances of avoiding errors related to casting.

### 4.9 Confusing `==` with `equals()`

In Java, confusion often arises about the issue of equality (Haggar, 2000). Such confusion exists with respect to the method `equals()` and operator `==`. The method
equals() performs a character-by-character comparison on two objects of type string and it can only be used with object references (e.g., string objects) and not primitive data type variables (e.g., int, double, etc.) (Daconta et al., 2000; Deitel & Deitel, 2002; Haggar, 2000). Operator == tests if two references are referring to the same object (Daconta et al., 2000; Deitel & Deitel, 2002; Haggar, 2000). Method equals() and operator == are, therefore, different and may produce different results. The example in table 18 has been adapted from Daconta et al., (2000) to illustrate such difference.

Table 18 – Method equals() vs Operator ==

<table>
<thead>
<tr>
<th>Method Call</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>s0.equals(s1)</td>
<td>true</td>
</tr>
<tr>
<td>s0.equals(s2)</td>
<td>true</td>
</tr>
<tr>
<td>s0 == s1</td>
<td>false</td>
</tr>
<tr>
<td>s0 == s2</td>
<td>true</td>
</tr>
</tbody>
</table>

In table 18, the String type references s0 and s1 refer to two different String type objects whose content is identical: Programming. The String type reference s2 refers to the same String type object as s0. This explains the results obtained in the output section of table 18:

```
...  
s0.equals(s2): true  
s0 == s1: false  
...  
```

Lack of awareness of the difference between method equals() and operator == may cause Java developers to commit errors which lead to incorrect results or unpredictable object behaviour (Deitel & Deitel, 2002). Such errors can be exposed by duplicating object comparisons using the == operator and equals() method and observing the results.
4.10 Issues with Inner Classes

Deitel & Deitel (2002) describe an inner class as a class which is defined in another class. The class where the inner class is defined is called the enclosing class (also known as the nesting or outer class). Inner classes are used mostly to implement event handling (Arnold et al., 2000; Deitel & Deitel, 2002). Arnold, Gosling, & Holmes define a few rules with respect to the access rights of inner classes, which if ignored can lead to errors. The first rule states that an inner class can access and modify all members (i.e. instance variables and methods) of the enclosing class. This can be achieved by accessing or modifying the required member directly with the need to qualify its name with the name of the enclosing class (Arnold et al., 2000; Deitel & Deitel, 2002). The second rule, is not commonly stated in Java references (Campione & Walrath, 1998; Deitel & Deitel, 2002; Horstmann, 2000; Hubbard, 1999; Liang, 2000; Rowe, 1998; Weber, 1996; Winder & Roberts, 2000). This rule states that:

"While an object of the inner class is always associated with an object of the enclosing class, the converse is not true. An object of the enclosing class need not have any inner class objects associated with it, or it could have many" (Arnold, Gosling, Holmes, 2000, p. 125).

The example shown in table 19 has been adopted from Roberts & Heller (1997) to illustrate the effects that multiple instances of the inner class can have on the enclosing class. This example features a class Outerone which contains an inner class called Innerone. Class Innerone contains a method called innerMethod() which accesses and modifies instance variable x of class Outerone. Class OuteronestTest instantiates multiple instances of the class Innerone.

Clearly, the example in table 19 is an embodiment of the two rules stated above. It also shows that multiple instances of the inner class are constructed (e.g. Innerone, innerTwo, etc.). Such instances have access to the enclosing class instance variables. If the code in the inner class (e.g. public void innerMethod()) is not carefully designed, it has the potential to change the enclosing class instance variables to values (e.g. int x) which may not comply with the specification, resulting in incorrect or invalid states of instances of the enclosing class. This implies that the values of the enclosing class instance variables need to be exposed using
getEnclosingClassInstanceVariable() methods. Lack of awareness of such rules can be the cause of errors when dealing with inner classes (Arnold et al., 2000; Roberts & Heller, 1997).

Table 19 - Error hazards with inner classes (Roberts & Heller, 1997)

```java
class OuterOne {
    private int x;
    public int getx() {return x;}
    public void outerMethod() {System.out.println("x is: " + x);}
    public class InnerOne {
        int y;
        public void innerMethod() {
            ++x;   // error hazard
            System.out.println("enclosing x is: " + x);
            System.out.println("y is: " + y);
        }
    }
}
class OuterOneTest {
    public static void main(String args[]) {
        OuterOne.InnerOne innerOne = new InnerOne();
        innerOne.innerMethod();
        OuterOne.InnerOne innerTwo = new InnerOne();
        innerTwo.innerMethod();
    }
}
```

Output:
enclosing x is: 1
y is 0
enclosing x is: 2
y = 0
x = 3

4.11 Memory Leaks

In Java, like in other languages, objects occupy memory. Such memory needs to be reclaimed and returned to the operating system when an object has served its purpose and is no longer needed. If memory is not reclaimed and returned to the operating system, the chances are that eventually there will be insufficient memory for the application to run, exhibiting poor software performance, software crashes or java.lang.OutOfMemoryErrors (Henry & Lycklama, 2000; Patrick, 2001). These situations are commonly known as memory leaks (Flanagan, 2001).

While in other languages (e.g., C++) developers are responsible for writing code that allocates memory to objects and reclaims it, in Java the task of reclaiming memory is
performed automatically. The utility program that performs such task is called a garbage collector (Nylund, 1999). A garbage collector runs automatically constantly searching for objects that are unreachable\(^1\) (i.e. no longer needed) by any other object in the software (Henry & Lycklama, 2000; Nylund, 1999). This has led to the misconceived popular belief that memory leaks are not possible in Java programs. Unfortunately this is not always the case (Nylund, 1999). The example in table 20 has been adapted from Flanagan, (2001) to illustrate the possibility of memory leaks in a Java program.

Table 20—Example of a possible Memory Leak

| class LeakExample { |
| public static void main(String args[]) { |
| int big_array[] = new int[100000]; |
| //this queries big_array and obtains a results |
| int result = compute(big_array); |
| //at this point big_array is no longer needed. However, |
| //it will be garbage collected only when the //method returns, because big_array is a local variable. |
| //The method, however, will never return, because, it |
| //loops infinitely, handling the user's input. |
| for (;;) {handle_input(result);} |
| } |

The example in table 20 shows the invocation of method compute(big_array) which queries big_array to obtain a result. After the result is obtained (i.e. after statement int result = compute(big_array); executes), big_array is no longer needed. Yet, big_array will still remain in memory and can only be garbage collected when the method (i.e. method main(String args[])) returns. This occurs because big_array is a local variable. Method main(String args[]), however, will never return because it is supposed to loop infinitely in order to handle the user’s input. This is a situation where an object (big_array) is left unused in memory. Situations like this are likely to

---

\(^1\) In this context, Begic (2001) recognises three types of objects, namely, reachable, resurrectable, and unreachable. Reachable objects are visible to the garbage collector, however, they are still being referred to by other objects in the software. The garbage collector will not attempt to clean such objects (Begic, 2001). Resurrectable objects are not referred to by other objects, but may become reachable when the garbage collector executes the objects’ finalize() method (Begic, 2001). Every object has a finalize() method which is guaranteed “to perform termination house keeping on the object just before the garbage collector reclaim the memory for the object” (Dethel & Dekel, 2002, p. 426). Unreachable objects are objects which cannot be reached and resurrected and they are the primary candidates to be garbage collected (Begic, 2001).
cause memory leaks, and cannot be resolved by the Java garbage collector (Flanagan, 2001).

Nylund (1999) provides three causes for memory leak errors. Firstly, unwanted object references can cause memory leaks. The easiest way to avoid these memory leaks is to identify objects that are no longer needed and assign their references to null. For example, in table 20, the statement: big_array = null; could be inserted after the result variable is initialised (i.e. after statement int result = compute(big_array); executes). The act of assigning null to a variable marks the object as able to be referenced by the variable for garbage collection (Deitel & Deitel, 2002).

According to Nylund (1999), a second cause for memory leaks is the developers’ failure to free native system resources. Native system resources are typically allocated through the Java Native Interface (JNI) by functions that are external to Java and are implemented in C or C++. For example, Java developers commonly use Abstract Windowing Toolkit (AWT) classes (e.g. Frame, graphics, etc.) and when such classes are no longer needed, they fail to release the system resources reserved for them using the dispose() or finalize() methods.

A third cause of memory leaks is when Java developers reuse third-party libraries (e.g. Java Development Kit (JDK), etc.) or use code developed by other developers. The reused code may already have errors due to the two causes discussed above (Nylund, 1999; Patrick, 2001). One way to avoid existing memory leaks in libraries is to become acquainted with them by checking errors that are published by the developers of such libraries. For example, Sun’s Java Developer Connection Bug Database publishes common errors in JDK libraries (Microsystems, 2002).

Patrick (2001) suggests that developers should carefully program Java software that includes structures that are likely to cause memory leaks. These include collection classes, such as hashtables, vectors, arrays, etc. (Flanagan, 2001; Patrick, 2001). In such cases, developers need to study the lifetime of objects of such classes and assign their...

---

13 As a matter of fact this must be done for all objects, in order to promote good practice (Flanagan, 2001; Nylund, 1999).
references to null when such objects are no longer needed (Flanagan, 2001; Haggar, 2000). A Java programmer may even explicitly call the garbage collector to execute using method system.gc()\(^3\).

Another structure that commonly introduces memory leaks includes event listeners (Henry & Lycklama, 2000; Patrick, 2001). Event listeners may cause memory leaks when an object is added to an event listener list, but not removed when the object’s usefulness has lapsed (Henry & Lycklama, 2000). Henry & Lycklama (2000) suggest that memory leaks involving lapsed listeners can be avoided by always pairing calls to addEventListener() and removeEventListener().

Patrick (2001) recognises that objects that are accessed from static structures are also prone to memory leaks. Static structures include instance variables, methods or even classes, which, once initialised, will stay in memory for as long as the program that defines them stays in memory (Nylund, 1999). This implies that any object that is referred to by a static structure will be kept in memory, although such an object may not actually be needed, thus, increasing the possibility for a memory leak (Nylund, 1999).

In order to avoid memory leak errors, Begic (2001) proposes the development of memory profiles of Java software that are suspected to result in memory leaks. A memory profile\(^4\) portrays the memory requirements of the software at various points during its execution (Begic, 2001). According to Begic (2001) the memory profile will help developers localise the memory “hotspots” and, thereby, make informed decisions on further steps to optimise memory consumption. Memory profiles may not only help prevent memory leak errors but may also expose their presence, if such errors have already been committed (Begic, 2001; Nylund, 1999).

\(^3\) However, calls to system.gc() do not guarantee the prompt execution of the garbage collector and the order in which any objects marked for garbage collection will be cleared. The call to System.gc() is just a suggestion to the Java Virtual Machine for the garbage collector to be called. This suggestion can be ignored (Delwel & Deitel, 2002).

\(^4\) Begic (2001) proposes the use of automatic tools such as Rational Purify in order to create memory consumption profiles. Other tools include JProbe, Optimizer, Insight, etc. (Henry & Lycklama, 2000; Nylund, 1999). Such tasks, however, can also be performed by either writing specialised classes (e.g. ObjectTracker class in Henry & Lycklama, 2000), by using operating system tools which help observe the memory needs of an application or process or by using Java Development Kit methods. For example, in class java.lang.Runt1me, methods freeMemory() and totalMemory() return the amount of unused and total memory, respectively.
Although Java, with the introduction of the automatic garbage collector has reduced the potential for memory leaks, such leaks have not been eliminated. Misconception about the capabilities of the garbage collector could itself be a cause for memory leaks (Flanagan, 2001; Henry & Lycklama, 2000; Nylund, 1999; Patrick, 2001). Memory leaks can be avoided when awareness about them and the structures that are prone to them is established.

4.12 Thread Deadlocks
Arnold, Gosling, & Holmes (2000) describe threads as the tool that the Java language provides to achieve the execution of multiple tasks concurrently. Campione & Walrath (1998) define a thread to be a “single sequential flow of control within a program” (p. 329). This “single sequential flow of control” may be responsible for a task that a program is expected to perform (e.g. see figure 1 (a)). The notion of a single thread, however, does not add any value to what is currently known about programs, because every program is a single sequential flow of control. The real value is added when multiple threads are used in a program to perform many tasks simultaneously (see figure 1 (b)). For example, within a browser, a user may scroll a page, download a file and play a sound concurrently (Campione & Walrath, 1998).

![Figure 1 - Threads](Campione & Walrath, 1998)
Sometime multiple threads may need to and attempt to access the same object concurrently. Daconta et al. (2000) illustrate this notion with an inventory e-commerce software example. Assuming that there are two threads (representing two independent customers) attempting to obtain the same last item in an inventory object simultaneously, both threads would have determined the availability of the item. However, it is possible that one thread would charge the customer and actually obtain the item, while the other thread would charge the customer and realize that the item has actually been sold out. The possibility of the occurrence of such a scenario requires the object (i.e. the inventory object) to indicate to the contending threads (i.e. the customer objects) whether it is being used by another thread. Such indication would prompt the contending threads to wait until the object (i.e. the inventory object) is free again (see figure 1 (c)). The mechanism that helps accomplish this is known as the Java monitor model and the object that different threads contend for is called a lock object (Deitel & Deitel, 2002; Holub, 1998a). While the example in figure 1 (c) shows that two threads are contention for the same object, in practice, two or more threads may be contention for two or more objects (Arnold et al., 2000; Campione & Walrath, 1998; Daconta et al., 2000; Holub, 1998a; Mitchell, 2000; Sun Microsystems & Services, 1998). When threads which attempt to use more than one object are not properly designed, they are likely to cause thread deadlocks (Arnold et al., 2000; Daconta et al., 2000; Holub, 1998a). Thread deadlocks may also be caused when Java developers possess inadequate knowledge about threads in general or about possible deadlock scenarios in particular (Arnold et al., 2000). Campione & Walrath (1998) have observed that:

"a deadlock occurs when two (or more) threads are each waiting for the other(s) to do something" (p. 354).

The scenario in figure 2 illustrates a thread deadlock situation.

---

35 The monitor is also known by the name of mutex; a name which was coined at the Digital Equipment Corporation (Holub, 1998a).
Figure 2 shows that Thread A will not release its hold on object X until it obtains Object Y. Likewise, Thread B will not release its hold on object Y until it obtains Object X. This scenario suggests the Threads A and B will be waiting for each other indefinitely, resulting in a deadlock. The symptoms of a thread deadlock occur when the application is 'stalled' or 'freezes' or 'hangs' (Arnold et al., 2000; Campione & Walrath, 1998; Daconta et al., 2000; Mitchell, 2000). Arnold, Gosling, & Holmes (2000) imply that thread deadlocks can be exposed by running the application many times. For instance, in figure 2, it is possible for Thread A to lock object X and object Y, before Thread B even starts its execution, in which case the deadlock will not occur. Alternatively, it is also possible for Thread B, to lock object Y and object X before Thread A starts its execution, in which case the deadlock will again not occur.

It is the responsibility of a utility program called the thread scheduler to determine which thread to start and when to start it. A Java developer does not have control over the thread scheduler. It is, therefore, important that any program suspected of ending up in a deadlock be run many times to trigger the presence of the deadlock (Arnold et al., 2000).
Arnold, Gosling, & Holmes (2000) suggest that it is the programmer's responsibility to prevent thread deadlocks from occurring. They propose a technique called resource ordering (Campione & Walrath, 1998; Holub, 1998b). With this technique, an ordering value is assigned to all objects whose locks must be acquired by a set of contending threads. It is the programmer's task to ensure that the locks on all objects are always acquired and released in that order. This ensures that:

"It is impossible for two threads to hold one lock each and be trying to acquire the lock held by the other - they must both request the locks in the same order so once one thread has the first lock, the second thread will block trying to acquire that lock, and then the first thread can safely acquire the second lock." (Arnold, Gosling, & Holmes, 2000, p. 254).

An example illustrating the resource ordering technique has been developed as a solution to a thread deadlock problem presented in Arnold et al., (2000). Due to the lengthy size of the code, this example has not been presented here. It can, however, be found in the following appendix A, section 1.2 where the student version of the Catalogue of Errors is provided. Other illustrated examples can also be found in Holub, (1998b).

5. References


Original and Revised Versions of the Catalogue of Errors

Table of Content

1. CATALOGUE OF REQUIREMENTS ERRORS 405
   1.1 OMISSIONS ................................................................. 405
   1.2 INCONSISTENCIES, AMBIGUITIES, AND INCORRECT FACTS .......... 408

2. CATALOGUE OF DESIGN ERRORS 413
   2.1 ALGORITHMIC ERRORS .................................................. 413
   2.2 INTERFACE ERRORS ...................................................... 416
   2.3 REUSE ERRORS ........................................................... 420
   2.4 STRONG COUPLING ERRORS .......................................... 425
   2.5 WEAK COHESION ERRORS ............................................. 431

3. CATALOGUE OF CODE ERRORS 436
   3.1 ABUSE OF NAMESPACE .................................................... 436
   3.2 FAILURE TO ESTABLISH CLASS INVARIANTS ......................... 439
   3.3 INCREMENT AND DECREMENT OPERATORS .............................. 442
   3.4 CONSTRUCTOR INVOKED OVERRIDDEN METHOD ....................... 444
   3.5 HONOURING CONTRACTS OF METHODS INHERITED BY DIFFERENT INTERFACES .......... 446
   3.6 OBJECT INITIALIZATION .............................................. 449
   3.7 ACCESSING METHODS VERSUS ACCESSING INSTANCE VARIABLES ...... 452
   3.8 CAST DOWN THE INHERITANCE HIERARCHY ............................ 455
   3.9 CONFUSING == WITH_EQUALS () ....................................... 458
   3.10 ISSUES WITH INNER CLASSES ....................................... 460
   3.11 MEMORY LEAKS .......................................................... 462
   3.12 THREAD DEADLOCKS .................................................. 466
1. Catalogue of Requirements Errors

1.1 Omissions

Original Version of the Catalogue of Requirements Errors

Error Name: Omission

The presence of omission errors in a requirements artifact indicates that a requirement is missing or that it has not been specified. The following is an example of an omission:

*Requirements Specification Text:* In the case of two incoming trains, each of which has access to a siding of adequate length, the slower one is to be routed onto the siding while the faster one continues on the main track.

*Omission Error:* There is no information about what to do if the two trains are moving at exactly the same speed as measured by the wayside location.

Severity

Critical Error.

Origin

Requirements Phase. Definition of Goals of Software.

Cause

i) Misinterpretation, misunderstanding or misconceptions of user communication or documentation.

ii) Inadequate knowledge, appreciation, or problem analysis

iii) Inadequate initial problem definition or requirement specification ambiguous or poorly communicated

iv) Problem with the way/process the requirements are specified

v) Lack of continual review of requirements during development

vi) Insufficient involvement of users in requirements specification

vii) Pure negligence from users or developers

Prevention

In order to prevent omission errors the following steps must be undertaken:

i) Identify software goals. A goal is an objective that must be achieved by the system under consideration. The rationale behind this integration is that the requirements are complete if they are sufficient to establish the goals they are refining. Goals can be identified by looking for keywords such as: objective, purpose, intent, concern, in order to, etc. in the user's statements or documentation.

ii) The second step constitutes the refinement of goals into subgoals. The subgoals are a lower level representation of goals whose accomplishment leads to the achievement of the goals from where they are refined. Goal refinement is intended to reduce the risk of incomplete requirements. For example, one of the goals of the San Francisco Bay Area Rapid Transit system is to serve more passengers. This goal can be satisfied by accomplishing either one of its two
subgoals, namely, *trains must be more closely spaced OR new tracks need to be added*. Upon completion of goal refinement, goals, subgoals (and, depending on the scope of the software, sub-subgoals, etc.), can be organised into a goal tree or matrix. The organisation of goals, subgoals, etc., into a tree or matrix shows not only how subgoals are derived, but also ensures the traceability of the goals to subgoals.

iii) The third step to prevent omissions is called goal conflict identification. It is possible that conflicts between/among goals may exist. A goal conflict is a situation whereby the accomplishment of one goal rules out the accomplishment of another. For example, in a scheduling software system, the goal of making timetables publicly available competes with security or privacy goals.

iv) The fourth step of preventing omissions requires the operationalisation of goals into requirements. Goal operationalisation entails the translation of goals into operational functions and constraints for the final requirements specification document.

v) Depending on the scope and the nature of the software being developed, the number of goals and requirements may grow considerably. This may exacerbate the organisation of goals and requirements and even encourage the introduction of omission errors. It is, therefore, necessary that goals and requirements be organised into traceability matrices (see e.g. Goals-Requirements Traceability Matrix). This is the fifth step of preventing omissions.

**Goal – Requirements Traceability Matrix**

<table>
<thead>
<tr>
<th>Requirements</th>
<th>Ensure Secure Transactions</th>
<th>Keep detailed transaction information</th>
<th>Etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validate User</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Validate Transaction</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Produce receipt</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Save transactions</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etc</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Symptom**

Omissions result in incomplete requirements artifacts. If omissions are not prevented or discovered early in software development, the resultant software will not solve the problem it was designed to address.

**Trigger**

- Involving the users of the software whose requirements are being developed to verify the developed requirements.
- Involving knowledge experts of the domain in which the software is being developed
- Using checklists.
Additions in the Revised Version of the Catalogue of Requirements

Errors

Example: Operationalisation of Goals into Requirements

The following ATM example can be used to illustrate the operationalisation of goals into requirements. The goal to,

Ensure secure Transactions

translates into the following operationalised constrained requirement:

Requirement: Validate User
Required Precondition: Valid ATM Card is used, Valid PIN is used
Required Postcondition: Valid ATM Card is used, Valid PIN is used

Example: Checklists

i) Are the described requirements sufficient to meet the system goals?
ii) Have all requirements been provided with the necessary inputs?
iii) Have all undesired systems states and events been considered, and the appropriate responses specified?

References

1.2 Inconsistencies, Ambiguities, and Incorrect Facts

Original Version of the Catalogue of Requirements Errors

Error Name: Inconsistency
An inconsistency error exists if information within one part of the requirements artifact (RA) disagrees or contradicts with other information in the same artifact. An example to illustrate an inconsistency error in a requirements artifact could be as follows:

Requirements text in section X of RA: ...an address must include a single unique four digit postcode integer...
Requirements text in section Y of RA: ...postcode 61512 is located in Western Australia...
Inconsistency Error: The 5 digit postcode integer in section Y of RSA is clearly inconsistent with the requirement in section X that postcodes should consist of four digits only.

Error Name: Ambiguity
An ambiguity error constitutes the unclear representation of a concept in a requirements artifact. An ambiguity error can be illustrated with an Air Traffic Controller (ATC) software system, which uses two aircraft location display formats. The small display format displays information about the flight carrier, number, altitude, heading, and destination. In order to reduce clutter, the large display format displays only the flight carrier and number information:

Requirements text: For up to 12 aircraft, small display format shall be used. Otherwise, the large display format shall be used.
Ambiguity Error: The ambiguity exists in “for up to 12”, which some people may interpret as “for up to and including 12” while others as “for up to and excluding 12”.

Error Name: Incorrect Fact
Incorrect fact errors are errors whereby a requirement asserts a fact that contradicts general domain knowledge or the conditions that have been specified for the software being developed. An example of an incorrect fact in the requirements specification artifact of a Traffic Light Control software system is as follows:

The user requires that: The traffic light should have the following sequence of colours: Green -> Amber -> Red -> Green
Requirements Specification Text: The traffic light should have the following sequence of colours: Green -> Red -> Green
Incorrect Fact Error: Clearly, the warning Amber has been omitted from the sequence of colours of the traffic light.

Severity
Critical Errors.
Origin

Cause
viii) Misinterpretation, misunderstanding or misconceptions of user communication or documentation
ix) Inadequate knowledge, appreciation, or problem analysis
x) Inadequate initial problem definition or requirement specification ambiguous or poorly communicated
xi) Problem with the way/process the requirements are specified
xii) Lack of continual review of requirements during development
xiii) Insufficient involvement of users in requirements specification
xiv) Pure negligence from users or developers

Prevention
In order to prevent inconsistency and ambiguity errors viewpoints and scenarios must be used as follows:
i) Develop viewpoints for the software whose requirements are being developed. Viewpoints constitute all end-users or other systems that are interfaced to the software system whose requirements are being specified. For example, in an ATM software system some viewpoints include the bank manager, the home customer, the foreign customer, security officer, etc. Each viewpoint has an associated set of functional and non-functional requirements and constraints. For example, in the ATM system, the bank manager requires transaction reports, a home customer requires cash withdrawals, deposits, balance inquiries, etc.

ii) Resolve possible conflicts between/among viewpoints. Sometimes, conflicts may exist between different viewpoints. For example, in the ATM system the security officer viewpoint requirement that software needs to be maintained regularly interferes with the availability requirement of the home customer and bank manager viewpoints. Conflicts need to be identified, compared and prioritised on the basis of importance, and resolved.

iii) Outline the chronological sequence of events (i.e. the scenarios) with which the viewpoints interact with other systems. The following is an example of an ATM software:
Example of Scenarios for the ATM software system

<table>
<thead>
<tr>
<th>Normal ATM Use Scenario</th>
<th>ATM Use Scenario with Exceptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. ATM asks user to insert card; user inserts cash card</td>
<td></td>
</tr>
<tr>
<td>2. ATM accepts the card and reads its ID number</td>
<td></td>
</tr>
<tr>
<td>3. ATM requests password; user enters '1234'</td>
<td></td>
</tr>
<tr>
<td>4. ATM verifies the card ID number and password with central computer; central computer IDs details</td>
<td></td>
</tr>
<tr>
<td>5. ATM asks user to select transaction type (deposit, withdrawal, etc.); user selects withdrawal</td>
<td></td>
</tr>
<tr>
<td>6. ATM asks for the amount of cash; user enters AUD100.00</td>
<td></td>
</tr>
<tr>
<td>7. ATM requests central computer's approval for requested amount</td>
<td></td>
</tr>
<tr>
<td>8. ATM dispenses cash and requests user to collect it; user collects cash</td>
<td></td>
</tr>
<tr>
<td>9. ATM asks whether user wants to continue; user indicates no</td>
<td></td>
</tr>
<tr>
<td>10. ATM prints receipt, ejects card and requests the user to collect them; user collects them</td>
<td></td>
</tr>
<tr>
<td>11. ATM asks user to insert card</td>
<td></td>
</tr>
<tr>
<td>1. ATM requests user to insert card; user inserts cash card</td>
<td></td>
</tr>
<tr>
<td>2. ATM accepts the card and reads its ID number</td>
<td></td>
</tr>
<tr>
<td>3. ATM requests password; user enters 9999</td>
<td></td>
</tr>
<tr>
<td>4. ATM verifies the card ID number and password with central computer; central computer rejects password as incorrect</td>
<td></td>
</tr>
<tr>
<td>5. ATM indicates incorrect password and requests user to re-enter it. User enters 1234, ...</td>
<td></td>
</tr>
<tr>
<td>6. ...</td>
<td></td>
</tr>
<tr>
<td>7. ...</td>
<td></td>
</tr>
</tbody>
</table>

iv) Ensure that all possible scenarios have been produced for all possible viewpoints and that any produced scenario belongs to a valid viewpoint. To achieve this, produce a Viewpoint-Scenario Traceability Matrix (see below).

### Viewpoint - Scenario Traceability Matrix

<table>
<thead>
<tr>
<th>Viewpoints</th>
<th>Normal ATM Use Scenario</th>
<th>ATM use scenario with exceptions</th>
<th>Etc.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bank manager</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Home customer</td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Foreign customer</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Security Officer</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Etc.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

In order to prevent incorrect fact errors a prototype should be developed. A prototype presents the users with a cheap and executable imitation of the required system where they can see directly whether the requirements provided by the prototype correctly fulfil their needs.

**Symptom**

**Inconsistency**

An inconsistency error may exist with respect to software behaviour, characteristics, terminology used in the requirements artifact and timing characteristics of sequences of functions or events.

**Ambiguity**

Ambiguity errors can cause confusion and misunderstanding and different interpretations about the actual software behaviours and characteristics.
**Incorrect Fact**

If incorrect fact errors are not prevented from being injected into requirements artifacts or uncovered early, developers might end up in producing software with incorrect features or features which the user does not need.

**Trigger**

One way to trigger inconsistencies is to review the requirements artifact with the objective of ensuring that all concepts, (i.e. goals, requirements, viewpoints, scenarios, and their likely relationships etc.) are consistently described.

In order to trigger the presence of ambiguity errors in a requirements artifact (RA), any statements in the RA which are not clear should be shortlisted. If two or more people are then asked to interpret such statements, and more than a single interpretation is obtained, the statements are ambiguous. Ambiguity errors can also be identified by ensuring glossaries of terms are included into the RA. In this way, terms in the body of the RA that do not conform with the definitions provided in the attached glossary must be changed or respecified.

In order to identify incorrect fact errors, the requirements artifact should be judged against some reference standard that is assumed to be infallible (e.g. an oracle). The standard often is a human expert or the user who can judge correctness based on his/her knowledge.

Another way to identify inconsistency, ambiguity, and incorrect fact errors is by using checklists.

**Additions in the Revised Version of the Catalogue of Requirements Errors**

**Example: Checklists to identify inconsistency, ambiguity, and incorrect fact errors**

<table>
<thead>
<tr>
<th>Inconsistency Errors</th>
<th>Ambiguity Errors</th>
<th>Incorrect Fact Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Are all requirements mutually consistent?</td>
<td>1) Are the individual requirements for all viewpoints stated so that they are discrete and unambiguous?</td>
<td>1) Are all the described functions correct?</td>
</tr>
<tr>
<td>2) Are all requirements consistent with the overall system, and the respective scenarios?</td>
<td>2) Are all scenarios described clearly and unambiguously?</td>
<td></td>
</tr>
<tr>
<td>3) Are all requirements consistent with the actual operating environment?</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
References


2. Catalogue of Design Errors

2.1 Algorithmic Errors

Original Version of the Catalogue of Design Errors

Error Name: Algorithmic Error
An algorithm consists of a set of well-defined instructions to carry out an operation. An algorithmic error consists of a flaw in the design of an algorithm, which should accomplish an operation, given certain input. For example, in the following algorithm, noHrs should be multiplied by hrRate, and not added to it.

```java
public float computeSalary(int noHrs, float hrRate) {
    salary = noHrs * hrRate; //algorithmic error
    return salary;
}
```

Severity
Serious Error.

Origin
Design Phase. Definition of Algorithms.

Cause
i) Algorithmic errors may result from inadequate understanding of the specification of the method or procedure they are written for.
ii) Algorithmic errors may also be caused by thin application domain knowledge and fluctuating or conflicting requirements.
iii) Sometimes developers reuse algorithms that have been written by others. In such cases, it is possible that either the incorrect algorithm is reused or that the correct one reused, but it is not properly understood.
iv) Algorithmic errors may even be due to clerical phenomena (e.g. performing the wrong computation, such as addition instead of multiplication etc.).

Prevention
In order to avoid algorithmic errors, developers should:

i) Adopt the appropriate algorithm design process, and
ii) Produce robust algorithmic designs.

Adopting the appropriate algorithm design process involves several stages:

i) Define the problem, which the algorithm is meant to solve. The problem definition should provide sufficient detail, including input, output, handling of possible errors, and any information that is not explicit in the user's documentation, etc.
ii) Name the algorithm clearly and unambiguously and after the operation it is meant to carry out.
iii) Define the data that the algorithm needs and decide how to organise this data into the appropriate data structure (e.g. array, vector, list, etc.).
iv) Design the algorithm. If predefined tested algorithms are available they can be reused or adopted to solve the problem at hand.

v) Check and fine-tune the algorithm to ensure that it solves the problem it was designed for. This should be done with respect to a number of criteria, namely, correctness, clarity, executeability, terminability, and efficiency. An algorithm is correct if its logic is correct. An algorithm is clear if it is simple and it contains precise instructions. An algorithm is executable if the steps it prescribed can actually be carried out in practice. An algorithm is terminating if eventually it terminates its own execution. Finally, an algorithm is efficient if it prescribes the best possible way to solve the problem in terms of the number of computations and number of variables used.

vi) Plan how the correctness of the algorithm will be validated (e.g., plan how the algorithm will be tested by producing test cases and expected results).

Robust design can be produced by incorporating pre- and post-conditions in algorithm design. While pre-conditions assert what should be true before an algorithm starts, post-conditions assert what should be true after the algorithm terminates. Pre-conditions should be ascertained at the beginning of a method or procedure, whereas post-conditions should be ascertained before the method or procedure exits. An algorithm should guarantee that its post-conditions are met, if its pre-conditions are met.

Symptom
If algorithmic errors are introduced, they may result in an unexpected behaviour or an incorrect result. Therefore, although, the input may be correct, the intended output may not occur because of the algorithmic error.

Trigger
In order to trigger possible errors, an algorithm must be assessed while bearing in mind the definition of the problem. This assessment should help alert the designer to the presence of injected algorithmic errors. If the result of the assessment indicates that the problem has not been adequately addressed, the process must restart. The process of algorithm assessment is also known as desk checking. Desk checking is the process of walking test cases through the algorithm, and keeping a step-by-step record of the content of the various variables involved as the test cases pass through the algorithm logic in order to obtain an actual result(s) for each test case. Discrepancies between actual and expected results suggest the presence of algorithmic errors.

Additions in the Revised Version of the Catalogue of Design Errors

Example: Identifying Algorithmic Errors
Assume that class Employee contains a private attribute named salary and a public method named computeSalary defined as below:

```java
public float computeSalary(int noHrs, float hrRate) {
    salary = -noHrs * hrRate; //possibility ONE
    //salary = noHrs + hrRate; //possibility TWO
    return salary;
}
```
Step 1. Establish (as strict as possible) Preconditions and PostConditions

- noHrs, hrRate, salary > 0 (strict preconditions and postconditions)
- Salary = noHrs * hrRate; (very strict postconditions)

Step 2. Desk check against specifications list:

<table>
<thead>
<tr>
<th>Statement</th>
<th>noHrs</th>
<th>hrRate</th>
<th>Salary (expected)</th>
<th>Salary (actual)</th>
</tr>
</thead>
<tbody>
<tr>
<td>First Pass (possibility ONE)</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>-4 (error detected)</td>
</tr>
<tr>
<td>Salary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N pass ... (possibility ONE)</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>-20 (error detected)</td>
</tr>
<tr>
<td>First Pass (possibility TWO)</td>
<td>2</td>
<td>2</td>
<td>4</td>
<td>4 (error not detected)</td>
</tr>
<tr>
<td>Salary</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>N pass (possibility TWO)</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>12 (error detected)</td>
</tr>
</tbody>
</table>

References


2.2 Interface Errors

Original Version of the Catalogue of Design Errors

Error Name: Interface Error

Interface errors stem from the way objects interact in an application. Object interaction is based on the way an object is structured. The structure of every object includes a provides section, representing the services that the object offers, and a requires section, representing the services that the object will use during its operation. Two objects interact by one object (the client) calling the required services from another object (the server). If each object's behaviour is assumed to be correct while in isolation, interface errors can occur in two possible cases:

i) The client object does not call the server's services in a "legal" manner (i.e. does not provide the correct types of input arguments); and,

ii) The client does not interpret the server's correct answers (i.e. returned values) properly.

Severity

Serious Error.

Origin


Cause

A number of different conditions may cause the introduction of interface errors.

i) Firstly, developer's failure to understand that individual correct objects do not necessarily mean correct interactions as well.

ii) Secondly, sometimes developers do not adopt systematic plans to integrate individually correct objects.

iii) Another cause may be due to the developer's failure to account for likely polymorphic substitutions during object interactions.

Prevention

In order to prevent interface errors the following steps must be adhered by:

i) The first step to prevent interface errors from occurring is to identify threads of execution in the design artifact of the proposed software. A thread of execution constitutes a scenario of normal or exceptional usage. A thread describes an aspect of the behaviour of the required system, whose enactment is the result of the interaction of some objects of the system. While such interaction requires some objects to be clients, and others servers, it is underlaid by a sequential chain of method executions.

ii) The second step requires that pairs of client and server objects be identified for each interaction of the sequence underlying the identified thread. Each client/server pair needs to be analysed in terms of the number, correctness of order and type, and possible values of the parameters involved in the interaction.

iii) The third step is an extension of the second step and requires further analysis of client/server pairs for situations when polymorphic substitutions are involved. This is particularly important if the client and server belong to inheritance...
hierarchies. For example, run-time instances of the subclasses of the client may act as polymorphic substitutes for an instance of the client in its interaction with a server object. Similarly, run-time instances of the subclasses of the server may act as polymorphic substitutes for an instance of the server in its interaction with a client object. Depending on the depth of the inheritance hierarchies where the client and server objects are located, polymorphism may compound the number of possible interactions.

When client and server objects interact, at least one method is invoked. The invoked method may have parameters and may return a result. In this context, polymorphism should also be considered as a compounding factor for the possible number of interactions when a client and server object interact by passing parameters that are instances of other objects, which are henceforth referred to as object parameters. If the object parameters belong to inheritance hierarchies, the polymorphic substitutions by the subclasses or ancestor classes of the object parameters must also be considered. All possible interactions that may occur due to polymorphic substitutions must be identified and appropriately validated.

Symptom
If interface errors are introduced in design artifacts, they may be observed in the form of a client object interacting with a server object attempting to use the incorrect or inexistent method. This means that a method may be attempted to be used with the wrong number, type and order of parameters supplied to it.

Trigger
In order to identify interface errors, developers should analyse and validate design artifacts by examining client and server objects and ensuring that they interact via the correct method invocations. In addition, developers should also check that the appropriate method is called with the correct type, number and order of parameters. Sometimes, interface errors introduced in a design artifact may escape to the coding phase. In such cases, besides code analysis and validation checks, developers may use tools that perform these checks automatically (e.g., compilers, etc.). However, as languages become more dynamic these checks are not always successful. The following example illustrates this point:

```java
String className = "VETOABLECHANGELISTENER";
Class parameterTypes[] = {Class.forName(className)};
```

In this example no automatic tool can check the correctness of the `className` `String`. If `className` is misspelt the error may only be flagged at run-time.
Additions in the Revised Version of the Catalogue of Design Errors

Example: Interface Error

Requirement:
"At the end of the month the payroll for all employees in the company is taken"

Scenario:
1. Examine employee n
   1.1 Count no of hours employee n has worked
   1.2 Retrieve hourly rate employee n is rewarded with
   1.3 Salary of employee n is the product of 1.1 and 1.2 above
   1.4 Add salary to total
   1.5 Repeat process for all employees in payroll

The above scenario requires the following classes:

employee class {computesalary(), Input: int noHrs, float hrRate, Output: float salary}
Payroll class {computeTotalsalaries(), Input: Employee, Output: float total}

In order to avoid interface errors the following conditions must be satisfied:

- The computesalary() method of the server object (Employee) provides the right parameters (int noHrs and float hrRate, two parameters, noHrs first hrRate second) to the method computeTotalsalary() of the client object (Payroll).
- The value (float) that is returned by the method computesalary() of the server object (Employee) is stored in the right type of variable (total of type float) in method computeTotalsalaries() of client object (Payroll).

If the above conditions are not satisfied, interface errors will occur as follows:

class Employee { //instances of this class are SERVER OBJECTS with respect to Payroll objects
    float salary;
    public float computesalary(int noHrs, float hrRate) {
        salary = noHrs*hrRate;
    }
}  
class Payroll { //instances of this class are CLIENT OBJECTS with respect to Employee objects
    Employee emp_1; //only one employee declared for simplicity
    int noHrs = 20;
    float hrRate = 2.0;
    float total;
    
    public float computeTotalsalaries(Employee emp) {
        total = total + emp.computesalary(hrRate, noHrs);
        //Interface breached, wrong order of parameters
        //total = total + emp.computesalary(noHrs, hrRate);
        //if total's type is different from method computesalary()'s return type,
//interface is breached
}

References


2.3 Reuse Errors

Original Version of the Catalogue of Design Errors

Error Name: Reuse Error
When one or more classes that belong to a library or an application are reused in a context which is different from the one they were originally designed for, reuse errors may occur. There are two types of reuse, namely, as-is reuse and specialisation or evolutionary reuse.

'As-is' reuse occurs when the to be reused class exists and possesses the required functionality. Instances of such classes may be used directly or they may be incorporated as part of the implementation of other classes. This type of reuse is considered to be safe and successful because of encapsulation and information hiding. Encapsulation and information hiding prevent instances of the reused class from interfering with instances of the other reusing classes. Nevertheless, assuming that the reused class is correct, the relationships between the reused class or any class that can polymorphically substitute it (e.g. classes in the same inheritance hierarchy), on one hand, and the interacting class or any class that can polymorphically substitute it, on the other hand, must be considered for testing.

Specialisation or evolutionary reuse occurs when an existing class provides similar, but not the exact kind of functionality that is required by the proposed software. In this case, a new class can be incrementally developed by inheriting from the existing class. The new class can then locally define new instance variables or methods as necessary. Evolutionary reuse introduces new and often complex dependencies between reused and new code, which defeats the purpose of encapsulation, and thus, provides an alternative for reuse errors to be introduced. In this case, reuse errors are favoured by the object-oriented feature of inheritance which allows a subclass in an inheritance hierarchy to directly access and use instance variables and methods defined in its superclass. In such cases, the relationship between the new subclass and the reused superclass must be thoroughly tested. The following example shows how specialisation or evolutionary reuse can cause errors.

```java
class Baseclass {
    int inherited(int x) {
        int redefined() { // returns a an int range [1..10]
    }
}

class DerivedClass extends Baseclass {
    int redefined() { //returns a an int range [0..20]
        int inherited(int x) { //inherited from the base class
    }
}
```

If method `inherited(int x)` contains the following algorithm:

```java
if(x<0) {
    x = x/redefined();
    return x;
}
```
In this case, if method `inherited(int x)` is called with regard to a subclass object `derivedClass`, it is possible for a division by zero error to occur.

Severity
Serious Error.

Origin
Design Phase. Inheritance Definition.

Causes
The following are the most frequently cited causes underlying reuse errors:
i) Typically, reuse errors occur when developers fail to understand the possible ways in which the properties of the superclass class interact with the properties of a subclass. This cause can be particularly problematic when the source code of the superclass to be reused is not available and its documentation is not sufficiently detailed.

ii) Another possible cause for reuse errors is when developers possess a poor understanding of dependencies that are introduced due to inheritance or when developers possess a poor understanding of the to be reused classes and the hierarchies that they belong to and when they are unable to use polymorphism properly.

iii) External pressure on developers (e.g. managerial influence) to adopt reuse can be another cause for reuse errors.

iv) Another reason why reuse errors may be introduced is that developers do not adopt a systematic reuse process.

Prevention
In order to avoid reuse errors, developers need to identify and evaluate the usage context. This requires the following steps:

i) identifying the class to be reused and learning more about its functionality by trialing it with a driver program in example applications,

ii) assessing similarities between what is offered by the to be reused class and what is required by the application, and assessing where the to be reused class will be reused,

iii) studying other aspects of the class to be reused and its impact on the required functionality. It is important that the to be reused class is domain independent and sufficiently abstract to fit into the new environment domain,

iv) the next step requires the developer to determine whether the class is to be reused on an 'as-is' basis or whether further specialisation is required, and

v) the final step requires that, once the reused class is incorporated into the new application, immediate analysis and testing and debugging ensues.

Symptom
If reuse errors are actually introduced, their symptoms can be varied and will typically range from unexpected and undesired application behaviour to unknown and unpredictable side effects. Developers are, therefore, better off preventing them than removing them.
Trigger
Reuse errors can be triggered by reviewing design artifacts and ensuring the following:

i) No errors occur when subclass methods interact with superclass instance
variables or methods; and
ii) No errors occur when superclass methods are reused in a subclass
environment.

Additions in the Revised Version of the Catalogue of Design Errors

Example: Reuse Error
The following example shows the circumstances in which reuse errors may occur.

class Employee {
    protected int noHrs;
    protected int eType;
    protected int salary; //type chosen for simplicity
    public int hourlyRate(int eType) {
        if (eType==1) {return 1;}
        else if (eType==2) {return 2;}
    }
    protected int calculateTotalHrs() {
        if (salary>0) {
            noHrs = salary/(hourlyRate(eType)); //OOPS
            return noHrs;
        }
    }
}
class SalesPerson {
    public SalesPerson(int empType, int sal) {
        eType = empType;
        salary = sal;
    }
    public int hourlyRate(int eType) {
        if (eType==1){return 0;}
        else if (eType==2) {return 1;}
    }
}

class Test {
    public static void main(String args[]) {
        SalesPerson sp = new SalesPerson(1, 10000);
        sp.calculateTotalHrs(); //division by 0 exception will
        //be raised
    }
}

Class to be reused: Employee
Reusing class: SalesPerson
Method to be reused: calculateTotalHrs()
Existing environment of Method to be reused:
- Input:
  - salary with domain [0..32676];
  - hrRate as return by method (hourlyRate(eType)) with domain [1,2];
- Output:
  - noHrs with domain [0..32676]
New environment of Method to be reused:
- Input:
  - salary with domain [0..32676];
  - hrRate as return by method (hourlyRate(eType)) with domain [0,1];
Compare and contrast between the two environments:

- It is possible that in the new environment (salesPerson) method `hourlyRate()` may return value 0, which is not specified in the existing environment of the `calculateTota1Hrs()` method. Since method `hourlyRate(eType)` is being invoked from the denominator (see method `calculateTotalHrs()`) there exists a possibility that a division by zero exception may be raised.
- This above indicates the existing code does not provide for a possible division by zero exception to be handled.

References


2.4 Strong Coupling Errors

Original Version of the Catalogue of Design Errors

Error Name: Strong Coupling Error
Coupling describes the interdependencies between different objects of a software system. Interdependencies between objects exist due to the interaction between client and server objects. Good software design exhibits low or weak coupling between objects as described by simple explicit interrelationships between objects. Strong coupling between objects makes software designs prone to errors and encourages error propagation. Strong coupling must therefore be avoided.

Severity
Serious Error.

Origin
Design Phase. Inheritance, Aggregation, and Association Definition. Identification of Interactions.

Causes
In general, there is agreement in the literature that failure to understand and recognise the different types of strong and weak coupling may diminish the chances of developers actually using weak coupling and avoiding strong coupling in their designs.

Prevention
Many suggest that a simple count of the number of objects that any given object interacts with is sufficient to define coupling between objects. Others, however, argue that more fine-grained descriptions capturing the different types of interactions between objects provide more precise guidance to software designers who wish to produce loosely coupled designs. With more fine-grained levels of coupling, acceptable and unacceptable levels can be recognised. Developers can recognise what types of coupling should be used and what types should be avoided. In general, three different types of coupling, namely, interaction coupling, component coupling, and inheritance coupling are recognised. Each type of coupling is comprised of its own acceptable and unacceptable coupling levels.

Interaction Coupling
Interaction coupling occurs when methods invoke each other. Such invocations may involve data sharing and may occur across different types of objects. There are five levels of interaction coupling ranging from content to data coupling:

i) Content coupling is the strongest and worst form of interaction coupling and occurs when a method can directly access the implementation of another;

ii) External coupling occurs when methods communicate via a structured global shared data space. For example, methods share the public instance variable of a class:
class Employee {
    double currentTax;
    double yearToDateTax;
    public void calculateTax(Income i) {
        if (i<=6000.0) {
            currentTax = 0;
        } else if (i< 20000.0 & i>=6000.0) {
            currentTax = i*0.17;
        } else if (...) {
            ...
        }
    }
    public void calculateCumulativeTax() {
        yearToDateTax = yearToDateTax + currentTax;
    }
} //external coupling example

iii) Control coupling occurs when one method passes data to a second method that dictates what the second method should or should not do. The following example illustrates control coupling:

class Tax {
    double totalTax;
    double currentTax;
    public void applyTax(double yearlyIncome) {
        switch (yearlyIncome) {
            case income1: totalTax = totalTax + 0.4*yearlyIncome;
                            break;
            case income2: totalTax = totalTax + 0.5*yearlyIncome;
                            break;
            default: totalTax = totalTax + 0.3*yearlyIncome;
        }
    }
}

class Employee {
    Tax taxPayable;
    double totalIncomePerYear;
    int incomeBracket;
    public void calculateTax() {
        totalIncomePerYear = calculateTotalIncomePerYear();
        taxPayable.applyTax(double totalIncomePerYear);
    }
    private double calculateTotalIncomePerYear() {
        //code to calculate total income;
    }
}

The weakness of control coupling is that the passing of the control attribute between the methods (totalIncomePerYear) implies that one method (calculateTax()) controls the internal logic of the other (applyTax()). Control coupling is not prohibited by object orientation but it should be avoided, since the change of the implementation of a method may cause hidden changes to the behaviour of the control coupled methods.
For example, if totalIncomePerYear instance variable is changed from a primitive data type into the wrapper class Double, any change in the implementation of Employee, should have its appropriate reflective change made in class Tax.

If, however, a hierarchy of Income objects is created each of which has its own specialised returnTax() method, then the implementation of the applyTax() method could be simplified considerably, but passing to it an Income object and calling the Income object's returnTax() method. Polymorphism and dynamic binding will kick in and perform a run-time determination of actual type of the Income object passed in and the appropriate returnTax() method to be called.

iv) Stamp coupling occurs when two methods communicate via the parameter passing of data structures, however, the called method uses only selected parts of the data structure that it receives. The following example, illustrates stamp coupling:

```java
class Employee {
    double sales;
}
class SalesStatistics {
    double accumulatedSales;
    public void addsale(Employee e) { //e is an instance of Employee class
    }
}
```

The method addsale() should not take an Employee object as a parameter, which leads to stamp coupling, but the value of the relevant instance variable, i.e. sales of a particular employee, which should lead to data coupling.

v) Data coupling occurs when two methods communicate only by passing relevant parameters. This is the best form of coupling. The following example illustrates data coupling:

```java
class Employee {
    double sales;
    public double getSales() {return sales;}
}
class SalesStatistics {
    double accumulatedSales;
    public void addsale(double sale) { //sale can be obtained by invoking
        //EmployeeObj.getSales()
    }
}
```
Content and control coupling are considered unacceptable, and should, therefore, be avoided. In general, external and stamp coupling are unacceptable, but exceptions are recognised. For example, for two methods that are externally coupled, their coupling is acceptable if the global data that the methods share is read-only (i.e. a set of constants). For two methods that are stamp coupled, their coupling is acceptable if one method uses all of the elements of the data structure that is passed in as a parameter by the other method. The only universally acceptable form of coupling is data coupling.

Component Coupling
Component coupling is another type of coupling. While interaction coupling concerns interactions between methods in the same or different classes, component coupling refers to situations whereby an object of one class is used as:

(i) an instance variable within another class,
(ii) as a parameter of a method of another class,
(iii) as local variable of a method of another class, or
(iv) as a parameter of a method which is invoked from within a method of another class.

In the above list of component interactions, (i) and (ii) are also known as specified coupling, because coupling between object or components is clearly specified in the interface of the class. Whereas, (iii) and (iv) are also known as scattered coupling, because the entire body of the class has to be closely searched for this type of coupling to be identified. The strongest form of component coupling is called hidden coupling. Hidden coupling occurs when the interaction between two instances of two classes is implicit. For example, cascading method invocations, like the method calls in the statement:

```java
getContentPane().add(new Label("MyLabel"));
```

constitute hidden coupling between an instance of JApplet where `getContentPane()` is called from, and the instance of class `Container`, which is returned by `getContentPane()`. Hidden coupling must be changed into at least scattered coupling. For example, the above line of code can be changed into the following:

```java
Container c = this.getContentPane();
Label l = new Label("MyLabel");
c.add(l);
```

The weakest form of interaction coupling exists when there is no direct coupling between the classes. Although, no direct coupling is referred as the theoretical optimum of component coupling, in practice no direct coupling may not be always possible to achieve, because normally objects from different classes are expected to interact with each other. Normally, specified and scattered component coupling are acceptable, whereas hidden coupling, being the worst type of coupling, must be avoided or changed into scattered coupling.

Inheritance Coupling
Inheritance coupling concerns objects of classes whereby one class is the superclass and the other is the subclass. Four levels of inheritance coupling are recognised. They range from modification to no inheritance coupling at all:
i) Modification coupling is the strongest and also the worst form of inheritance coupling. Modification coupling occurs when subclasses arbitrarily and unrestrictedly change or even delete methods inherited from the superclass. Two types of modification coupling are recognised:

a) Method signature modification coupling, which occurs when, not only the implementation, but also the signature of the inherited method is changed; and,

b) Method implementation modification, which occurs when only the implementation of the method is changed without any restriction.

ii) Refinement coupling is similar to modification coupling, however, it is weaker. This is because subclasses can only change inherited methods as restricted by predefined rules. Two types of refinement coupling are recognised, namely:

a) Method signature refinement coupling occurs when the subclass refines the signature and the implementation of at least an inherited method, without violating the semantics of that method. There are two rules that prevent the violation of method semantics after the subclass refines it. The covariant rule states that the return types (e.g. class C) of a refined method may be replaced by subclasses of that type (e.g. class C', which is a subclass of C). The contravariant rule augments what the covariant rules says with the additional condition that the types (e.g. class D') of the input parameters of the refined method may be replaced by supertypes (e.g. class D, which is a superclass of D').

b) Method implementation refinement coupling occurs if the refined method in the subclass reuses the code of the superclass method it overrides, in addition to any extra refinement code that it may introduce.

iii) Extension coupling is the best form of inheritance coupling, which occurs when the subclass adds new methods and attributes without modifying or refining either the signature or the implementation of any of the inherited methods.

iv) No inheritance coupling occurs when there is no inheritance relationship between the classes.

The only form of inheritance coupling that is acceptable is extension coupling (apart from no inheritance coupling) and other levels of coupling should be avoided.

Symptom

Strong or excessive coupling increases the complexity of the interdependencies between objects making the resulting software harder to understand, change, or correct, while favouring error propagation between objects. Other implications resulting from the presence of unacceptable levels of coupling in a design include, an increased susceptibility to errors of the affected sections of the design.

Trigger

In general, it is suggested that unacceptable coupling levels can be detected by using reviews and inspection of design artifacts. If the presence of such levels of coupling is detected in a design artifact, the redesign of the artifact is warranted, and this is always worse than avoiding unacceptable levels of coupling in the first place.
Additions in the Revised Version of the Catalogue of Design Errors

Examples: None

References


2.5 Weak Cohesion Errors

Original Version of the Catalogue of Design Errors

Error Name: Weak Cohesion Error
Cohesion describes the relatedness or connectivity of the different elements (i.e. instance variables and methods within a class or statements within a method) that a designer includes in the same class. If all elements of the class serve the same goal of the problem domain, the class has a strong cohesion, whereas, if the elements of the class are unrelated, the class has a weak (poor or low) cohesion. Strong cohesion is desirable, whereas weak cohesion is not.

Severity
Serious Error.

Origin

Causes
Although in practice it may not be necessary to determine precise cohesion levels, failure to understand and recognise what is cohesion is acceptable (i.e. strong) and unacceptable (i.e. weak) may cause developers to design non-single-minded classes and methods.

Prevention
In order to prevent weak cohesion errors, developers must be able to recognise the different types and levels of cohesion and determine which ones are acceptable and which ones are not. In general, three levels of cohesion are recognised, namely, method cohesion, class cohesion, and inheritance cohesion. Each type of cohesion is further broken down into levels.

Method Cohesion
Method cohesion shows how the statements included into a method are related. Method cohesion ranges from coincidental to functional cohesion and is based on Myer's (1978) classical definitions of cohesion:

i) Coincidental cohesion is the weakest form of cohesion indicating that all the statements that are carried out in the method have nothing in common besides the fact that they reside in the same method.

ii) Logical cohesion methods include statements with similar functionality that are included together not because they are related, but because they happen to perform similar functions. For example, a method called `inputAll()` that inputs customer names, employee time-card information, inventory data, etc. would be an example of a method with logical cohesion.

iii) Temporal cohesion methods include statements that are combined into a method because they are performed at the same time. For example, a `startup()` method might read a configuration file, initialise a scratch
file, set up a memory manager, and show an initial screen. The statements that perform such functionality are placed together simply because they are required at the same time.

iv) Procedural cohesion methods include statements that are connected by the same control flow, i.e. carried out in a specified order. For example, if the users of a reporting system like reports to be printed out in a given order, method print\text{All}\() may include statements that print a revenue report, an expense report, a list of employee phone numbers, and invitations to a party.

v) Communicational cohesion methods include statements that make use of the same data but are not otherwise related. For example, method getName\text{AndChangePhoneNumber}(\text{Employee } e) has communicational cohesion if both the name and the phone number are stored in an Employee object.

vi) Sequential cohesion methods contain statements that are performed in a specific order and share data from step to step. For example, method getFile\text{DataAndPerformComputations}\() would have sequential cohesion, because it would sequentially carry out three different steps, namely, open file, read file, perform computations on the same file.

vii) Functional cohesion is the strongest form of cohesion where methods possess sequential cohesion and all statements contribute to a single task or objective. Examples of strong cohesion methods include cos\(),\ get\text{CustomerName}\(),\ deleteFile\(),\ computeLoanPayment\(),\ etc.

Although functional cohesion is the best form of cohesion in practice, it is not easy. In fact, it is nearly impossible to develop methods with functional cohesion. In this context, methods with functional, sequential, communicational, and temporal cohesion are acceptable in practice. Methods with procedural, logical and coincidental cohesion are unacceptable.

**Class Cohesion**

Class cohesion addresses the binding of elements within the same class. Here, the elements of a class are comprised of non-inherited methods and instance variables. Five levels of class cohesion are recognised:

i) Classes are said to have separable cohesion, if they contain methods that access none of the class's instance variables, nor invoke any other method within the class. In this case, the class represents multiple unrelated concepts. This is the weakest level of class cohesion. The following example illustrates separable cohesion:

class Employee {
    String name;
    String address;
    Date birthDate;
    Date hireDate;
    ...  
    void computeAge() {...}
    void computeSalary() {...}
    int computeCompanyRevenue(Project p[]) {...}
}
The method `computeCompanyRevenue()` takes all projects of a company as input parameters and computes the accumulated revenue of that company. It neither accesses any instance variable of `Employee` nor does it invoke any other methods of `Employee`. To improve cohesion the method `computeCompanyRevenue()` should be factored out and included into a different object class, e.g. `Company`.

i)

A multifaceted cohesion class represents multiple related concepts. In such a class at least one method references an instance variable or invokes methods of a different, but related concept of the same class. The following example illustrates a class with multifaceted cohesion:

```java
class Reorder {
    Item reorderedItem;
    Company reorderedForm;
    int discount;
    int quantity;

    public boolean expectedRevenue() {
        // Method implementation
    }
}
```

Method `expectedRevenue()` calculates the expected revenue by subtracting the discount given for a company from the price of an item and multiplying this difference by the quantity of reordered items. The discount given depends on the company. Therefore, the cohesion of the `Reorder` class can be improved by including `discount` into class `Company`:

```java
class Company {
    String companyName;
    double discount;

    double discount() {
        // Method implementation
    }
}
```

ii) A non-delegated cohesion class comprises instance variables that describe only part of the concept that the class is supposed to represent. The following example illustrates a class with non-delegated cohesion:

```java
class Employee {
    String name;
    Date birthdate;
    Project involvedInProject;
    Employee managerOfProject;

    public double computesalary() {
        // Method implementation
    }
    boolean managerIncomeHigherThanAverageInProject() {
        // Method implementation
    }
}
```

In class `Employee`, the instance variables `birthdate` and `involvedInProject` depend directly on the instance variable `name`. However, the instance variable `managerOfProject` depend directly by the project referred to by `involvedInProject` and indirectly on the name instance variable. This constitutes a non-delegated cohesion.

To improve the cohesion of class `Employee` the instance variables `managerOfProject` and method `managerIncomeHigherThanAverageInProject()` should be delegated to class `Project` as follows:
class Project {
    Employee managerOfProject;
    Date startDate;
    Date expectedEndDate;
    
    boolean managerIncomeHigherThanAverageInProject() {
    }
}

Concealed cohesion classes include instance variables and methods such that, if regrouped, might form a new distinct and useful class. The following example illustrates a class with concealed cohesion:

class Employee {
    String name;
    STRING JOBPROFILE;
    int dayOfBirth;
    int monthOfBirth;
    int yearOfBirth;
    int dayOfHire;
    int MONTHOFHIRE;
    int yearOfHire;
    
    }

The instance variables describing the various dates may be factored out to a new class with instance variables day, month and year. The respective instance variables of the class Employee are then replaced by two instance variables of type date, namely, birthDate and hireDate, as follows:

class Employee {
    Date birthDate, hireDate;
    
}

Model cohesion is recognised as the strongest level of class cohesion and represents a single, semantically meaningful concept without containing methods which should be delegated to other classes or attributes which can be factored out into separate classes.

Class designs where cohesion is separable, multifaceted, non-delegated and concealed are unacceptable, and should, therefore, be avoided. These classes result in designs that are difficult to understand and reuse and they should be replaced with classes with model cohesion.

Inheritance Cohesion
Inheritance cohesion portrays the binding of all elements in a class. Here class elements are comprised of all methods and instance variables of a class, i.e. inherited and non-inherited. Sometimes inheritance is used solely to avoid data and code duplication between otherwise conceptually unrelated classes. For example, in an Elevator Simulator example class Elevator and Floor are both shown to be subclasses of a class called Location. This is the weakest kind of inheritance cohesion and should, therefore, be avoided. The strongest form of inheritance cohesion occurs when subclass-
superclass relationships are dictated by conceptual relationships, i.e. inheritance is used to define specialised children classes. For example, in a university library system both classes Students and Lecturers are subclasses of class person.

Symptom
Classes where cohesion is unacceptable will not only result in designs which are poorly organised, hard to understand, debug, reuse and modify, but as the research shows, such designs are also subject to an increased susceptibility to errors (i.e. error proneness). If weak cohesion designs are produced, the best remedy is re-designing classes to make them more cohesive.

Trigger
In general, unacceptable types of cohesion (either method, class, or inheritance) can be detected by reviews and/or inspections of design artifacts.

Additions in the Revised Version of the Catalogue of Design Errors

Examples: None

References


3. Catalogue of Code Errors

3.1 Abuse of Namespaces

Original Version of the Catalogue of Code Errors

Error Name: Abuse of Namespaces Error
Java uses the notion of namespaces to identify and manage the different program construction elements. A program construction element is defined to represent an identifiable program part that serves a unique purpose. For example, a method is an identifiable part of a Java program and it serves the purpose of providing a given service. A package serves the purpose of grouping related classes together. Other program construction elements include classes, instance variables, local variables, and labels. The definitions of these program construction elements can be found in Deitel & Deitel (1999a). In this discussion, a program construction element will also be referred to as simply an 'element'.

Java recognises six different categories of namespaces corresponding to each program construction element, namely, package names, class names, instance variable names, method names, local variables names, and label names. In order to give programmers greater flexibility, different namespaces allow programmers to use the same name to represent different program construction elements. For example, the same name may be used to represent a particular instance variable and a method and the compiler uses the context in which a name is used and the element scope rules to determine the element that the name represents. Element scope rules control the accessibility of the element to other elements of the program. For example, in the statement: \texttt{c.v=3.14;} \texttt{v} can only represent an instance variable \texttt{c}, because a programmer cannot assign a value to a package, class, method, local variable or label.

The flexibility to assign identical names to different program construction elements may be abused by developers and result in errors. The example shown below illustrates instance variable hiding. The output of the example shows that the local variable answer has hidden the instance variable answer resulting in an undesired outcome.

```java
public class Poor {
    public string answer = "yes!"; //instance variable answer
    public void wantMoney() {
        string answer = "No!"; //local variable answer
        System.out.println("Would you like $1Million> "+answer);
    }
    public static void main(string args[]) {
        Poor p = new Poor();
        p.wantMoney();
    }
}
```

Output:
Would you like $1Million> No!

Severity
Medium Error
Origin
Code Phase. Definition of Data Types. Object Initialisation.

Causes
The flexibility that the Java language provides to assign identical names to different program construction elements may be abused by developers or sometimes developers may not even be aware of how namespacing works. Both of these situations may cause errors. Namespacing mechanisms can, therefore, sometimes be a pitfall in Java and commonly result in hidden instance variable problems.

Prevention
Awareness of the following conventions help to avoid likely problems that may be associated with instance variable hiding:

i) Class instance variables may be hidden by local variables of the same name. A local variable includes variables that are declared locally in a method and parameters that are passed in as arguments to the method.

ii) Class instance variables may be hidden by subclass instance variables of the same name. This also includes situations where instance variables are 'multiply inherited' from two or more interfaces or from a superclass and one or more interfaces.

These errors may be prevented either by changing the names of the concerned variables or by qualifying the instance variable that is hidden with the this or super keywords. In order to avoid hiding an instance variable by a local variable, the instance variable should be qualified with the this keyword (case i), above in Prevention section). For example, in the class definition of the example presented earlier (class Poor), method public void wantMoney(), could be modified as follows:

```
system.out.println("Would you like $1Million> "+this.answer);
```

In this case the instance variable answer rather than the local variable answer will be accessed.

In order to avoid hiding a superclass instance variable by a subclass instance variable, the superclass instance variable should be accessed via the super keyword (case ii) above). For example, if class Poor were to have a superclass, say Person, with an instance variable answer, and method public void wantMoney() of class Poor needed to access the superclass' answer, then method public void wantMoney() would have to be modified as follows:

```
system.out.println("Would you like $1Million> "+super.answer);
```

In this case, the superclass (i.e. Person) instance variable answer will be accessed.

Symptom
If abuse of namespace errors go undetected, unpredictable object states may occur which can produce undesired results.
Trigger
Incorrect values of instance variables may result from instance variable hiding. If abuse of namespase errors are committed, then developers can trigger the presence of such errors by identifying all variables (including instance variable and local variables) sharing the same name whose scopes overlap. The actual values of the affected instance variables can be exposed by using utility methods, such as getInstanceVariableName() and compared to expected instance variable values.

Additions in the Revised Version of the Catalogue of Code Errors

Example: Abuse of Namespaces errors.

class Point {
    int x=1; int y=1;
    public Point(int x, int y) {
        x = x; // local variable x, rather than instance variable x is //initialised
        y = y; // local variable y, rather than instance variable y is //initialised
    }
}

package Reuse;

class Reuse {
    Reuse (Reuse Reuse) {
        Reuse:
        for (; ;) {
            if (reuse.Reuse(Reuse)==Reuse)
                break Reuse;
        }
        return Reuse;
    }
} //Pathological abuse of namespaces. Yet the code compile
// successfully

References


3.2 Failure to Establish Class Invariants

Original Version of the Catalogue of Code Errors

Error Name: Failure to Establish Class Invariants Error
A class invariant is an assertion expressing conditions on the instance variables of a class which must be preserved during the lifetime of every object of that class. For example, in a BankAccount class which reports an account balance and keeps track of deposits and withdrawals made on BankAccount objects via instance variables depositsList and withdrawalsList, the invariant could be established via the following assertion:

\[ \text{balance} = \text{total depositsList} - \text{total withdrawalsList} \]

Failure to establish and preserve the class invariant signals anomalous class implementation. The establishment and preservation of class invariant not only apply to methods that exist in the class, but also to methods that might be added later, thereby allowing control of the future evolution of the class.

Severity
Medium Error

Origin
Code Phase. Definition of Data Types. Object Initialisation.

Causes
Class invariants are devised by the programmer to reflect the constraints that the requirements impose on the instance variables of the class. This implies that failure to understand such constraints and provide for them in the implementation of a class will cause class invariants errors.

Prevention
Invariants need to be evaluated at the time the creation of an object and at the entry and exit points of all methods in a class and must hold whenever methods get called. Class invariants, however, do not need to be enforced during the evaluation of private methods because the private methods eventually get called by public methods and the invariants do not have to hold true during the execution of public methods. In the case of inheritance hierarchies, subclasses may inherit their parent's invariants, they may also override them or even create new ones to suit their specifications. If a class invariant is violated, the program should stop with an error message or throw an exception. The following example illustrates the establishment of the invariant of class Date (method invariant()) and one possible way invariants can be evaluated (e.g. method nextday()).

class Date {
    int day, hour;
    public Date (int d, int h) { day = d; hour = h;
        System.out.println(invariant());  //Invariant check
    }
}
public int getDay() { return day; }
public int getHour() { return hour; }
public int nextDay() {
    System.out.println(invariant()); // invariant check
    day++;
    System.out.println(invariant()); // invariant check
    return day;
}
public boolean invariant() {
    return (1 <= day && day <= 31) && (0 <= hour && hour < 24);
} // invariant rule

public static void main(String args[]) {
    Date d = new Date(31, 16);
    d.nextDay();
    System.out.println("Day:" + d.getDay() + ", Hour:" +
    d.getHour());
}

Output:
true // invariant okay - from date() constructor
true // invariant okay - at start of nextDay()
false // invariant violated - at end of nextDay()  
Day: 32, Hour: 16 // Erroneous output

Symptom
As the example (class Date) shows, invariant violation results in incorrect instance variable values which consequently result in invalid or inconsistent object states. This may ultimately bring about incorrect program output or behaviour.

Trigger
This implies that in order to trigger errors that violate class invariants, a developer should attempt to exercise suspicious code with values that attempt to violate the constraint imposed by the requirements. In the example shown earlier (class Date), the result (false) generated after the invocation of method invariant() after statement day++; suggests that there is a flaw in the implementation of method nextDay(). This is because the invariant of object d, has been violated as observed by its invalid state (day: 32, hour: 16).

Additions in the Revised Version of the Catalogue of Code Errors

Example: Failure to Establish Class Invariants Error

class Purse {
    private int dollars;
    private int cents;
    public void insert(int dollars, int cents) { // code to insert
        public void remove(int dollars, int cents) { // code to remove
            public static void main(String args[]) {

                PURSE PURSE = NEW PURSE();
                purse.insert(0, 50);
                purse.remove(1000, 50); // this call has the potential to
                // violate the invariant of
                // class Purse.
            }
        }
    }
}
References


3.3 Increment and Decrement Operators

Original Version of the Catalogue of Code Errors

Error Name: Increment and Decrement Operator Error
Java applies increment (++) and decrement (--) operators to numeric variables or numeric array elements. These operators are otherwise called unary operators because they are associated with a single operand. The increment or decrement operators increment or decrement the value of the operand by 1, hence the name. If an increment/decrement operator is located before the variable, it is called pre-increment/decrement operator, respectively. If an increment/decrement operator is located after the variable it is called post-increment/decrement operator. The following table summarises the differences between pre-, post- increment/decrement operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>Preincrement</td>
<td>++x</td>
<td>After variable x is incremented by 1, the new value of x is used in the expression where x is located.</td>
</tr>
<tr>
<td>++</td>
<td>Postincrement</td>
<td>x++</td>
<td>The existing value of x is used in the expression where x is located first, then x is incremented by 1.</td>
</tr>
<tr>
<td>--</td>
<td>Prededucrement</td>
<td>--y</td>
<td>After variable y is decremented by 1, the new value of y is used in the expression where y is located.</td>
</tr>
<tr>
<td>--</td>
<td>Postdecrement</td>
<td>y--</td>
<td>The existing value of y is used in the expression where y is located first, then y is decremented by 1.</td>
</tr>
</tbody>
</table>

In general, the expression x++ is equivalent to x = x + 1. However, there is a difference. In the former case, x is evaluated only once, whereas in the latter, x is evaluated twice. This is not always explained clearly in Java references. The following example illustrates a situation of the errors that are associated with increment and decrement operators.

class x {
    int i = 0;
    private int where() {return i++;}
    //public firstMethod() { ++arr[where()];...}
    //public secondMethod() { arr[where()] = arr[where()] + 1;...}
}

Assumptions: arr is an array of integers where all values are 0.

Output:
If firstMethod() is invoked, the second element of arr is incremented to 1.
If secondMethod() is invoked, the third element of arr is overwritten with 1.

Severity
Medium Error

Origin
Code Phase. Data Handling
Causes
In the literature it is suggested that failure to recognise the meaning of increment and decrement operators in general and the differences between expressions like \(x++\) and \(x = x + 1\) are likely to cause erroneous implementations.

Prevention
Errors associated with increment and decrement operators can occur if the rules associated with increment and decrement operators are violated or ignored. Therefore, they can be avoided by establishing awareness about such rules.

Symptom
Such erroneous implementations could consist of incorrect values for local, instance variables, etc.

Trigger
In order to expose possible errors associated with increment and decrement operators, developers should compare the results obtained from statements like \(\text{expression}++\) with the results obtained from statements like \(\text{expression} = \text{expression} + 1\).

Additions in the Revised Version of the Catalogue of Code Errors

Examples: None.

References
3.4 Constructor Invoked Overridden Method

Original Version of the Catalogue of Code Errors

Error Name: Constructor Invoked Overridden Method Error

In Java, a constructor is a special kind of method that is used to initialise objects of a given class. It is possible that a constructor might call methods of its own to carry out object initialisation. For example, the constructor `public C() {/*some implementation*/ m();}` of class `C` may call method `public void m(){/*m implementation*/}` of class `C` to carry out a given initialisation task. This practice is commonly used and causes no problems. However, non-intuitive incorrect results may be produced, if a new subclass of class `C`, say class `C1`, overrides the method `public void m(){/*subclass implementation*/}` that is being called from the `C1` superclass (i.e. class `C`) constructor. The following example illustrates this situation.

class super {
    super() { printThree();
    }
    void printThree() {System.out.println("Three");
    }
} //definition of superclass

class Test extends super {
    int indiana = (int)Math.PI;
    void printThree() {System.out.println(indiana);
    }

    public static void main(String args[]) {
        Test t = new Test();
        t.printThree();
    } //main method
} //definition of subclass

Output will be:

0
3

instead of the expected:

Three
3

This example features superclass `super` with constructor `super()` which calls method `printThree()`. Method `printThree()` is defined in class `super` and prints the value of string "Three". Class `Test` inherits from class `super`, but it overrides method `printThree()` to print the truncated value of `PI` (i.e. integer 3), which is stored in an instance variable called `indiana`. Method `printThree()` is called with reference to instance `t` of subclass `Test` and the result of its invocation is displayed. The results are not what one would normally expect.

The unexpected results occur because during the creation of the instance `t` of class `Test`, the `Test`'s superclass constructor (i.e. `super()`) is implicitly called first. Constructor `super()`, however, invokes the overriding rather than the original version of `printThree()` in class `super`. However, when the overriding version of `printThree()` is called, the instance variable `indiana` has not been initialised yet. This results in `indiana`'s default value (0) being retrieved by the overriding `printThree()`.
This phenomenon is also known as a downcall, because "the superclass "calls down to" the overridden method.

Severity
Medium Error

Origin
Code Phase. Object Initialisation.

Causes
One possible reason why constructor invoked overridden method errors occur is associated a with developer's failure to understand issues related to down calls and may be exacerbated when libraries of classes are developed from third parties and extended and their methods overridden.

Prevention
This type of error might not be very common, however, awareness of it can save considerable time if it is encountered.

Symptom
Errors associated with downcalls may exhibit non-termination or unexpected or incorrect behaviour, because when the superclass constructor calls down to an overridden method, the overriding subclass method may behave in an unexpected way.

Additions in the Revised Version of the Catalogue of Code Errors

Examples: None.

References

3.5 Honouring Contracts of Methods Inherited by Different Interfaces

**Original Version of the Catalogue of Code Errors**

**Error Name:** Honouring Contracts of Methods Inherited by Different Interfaces Error

Java allows a class to implement more than one interface. If two or more interfaces declare a constant with the same name or specify a method with the same signature, and the interfaces are then implemented by the same class, problems may occur in the implementing class. These problems are related to the fact that there is only one such constant or method whose implementation will ultimately have to be determined in the class which implements the interfaces. This situation is known as an interface name clash problem.

In this context, the real issue is whether a single implementation of the method can honour all the contracts implied by that method being part of different interfaces. The following example illustrates the interface name class problem.

```java
interface CardDealer {
    static final int variable1 = 5;
    void draw(); //flip top card
    void deal(); //distribute cards
    void shuffle();
}
interface GraphicalComponent {
    static final int variable1 = 4;
    void draw(); //render on default device
    void draw(Device d); //render on 'd'
    void rotate(int degrees);
    void fill(Color c);
}
class GraphicalCardDealer implements CardDealer, GraphicalComponent {
    int var1 = CardDealer.variable1;
    int var2 = GraphicalComponent.variable1;
    //draw(){ //which draw() will be implemented?
}
}
```

In the example shown above, it may be hard to implement method `draw()` in a class `GraphicalCardDealer` that honours both contracts suggested by the `CardDealer` and `GraphicalComponent` interfaces (i.e. flipping the card AND rendering on the default device) simultaneously. If a Java developer were to implement method `draw()` in a class `GraphicalCardDealer` to honour both contracts, then this method would exhibit weak cohesion. The `draw()` method, would therefore, be subject to drawbacks associated with weak cohesion.

**Severity**
Medium Error

**Origin**
Code Phase. Integration.
Causes

Errors associated with name clashes are normally caused when Java developers attempt to reuse classes from disjunct libraries to construct software.

Prevention

There are two ways that can be used to avoid name classes associated with issues in honouring contracts of methods inherited by different interfaces. Firstly, identical name constants declared in different interfaces can be accessed in a class which implements such interfaces, by fully qualifying their names. For example, in the class GraphicalCardDealer example, interface variables can be accessed as follows:

```java
int var1 = CardDealer.variable1;
int var2 = GraphicalComponent.variable1;
```

Secondly, if identical signature methods are involved in two (or more) interfaces, developers need to write a new interface which extends existing the ones (i.e. subinterfaces). In the subinterfaces, the names of the method(s) that is likely to clash should be changed. For example, the name class problem shown in the class GraphicalCardDealer example can be resolved as follows:

```java
interface MyCardDealer extends CardDealer {
    void drawAsInFlipTopCard();
}
class GraphicalCardDealer implements MyCardDealer, GraphicalComponent {
    void drawAsInFlipTopCard(); //Flip top card
    void draw(); //render on default device
}
```

Clearly, the interface MyCardDealer has been created, which inherits from the CardDealer. The MyCardDealer interface declares a new method, namely, void drawAsInFlipTopCard(), whose objective is identical to CardDealer's draw() method. However, here the name clash problem has been resolved, because, class GraphicalCardDealer can implement MyCardDealer and GraphicalComponent without any method name clash occurring.

This approach, however, can be problematic, because in Java, interfaces can be used as a reference to objects. If a CardDealer reference is used, it can only have access to the original CardDealer's draw() and not to the drawAsInFlipTopCard() of the MyCarddealer. This occurs because any object which is referenced through the CardDealer interface can only call methods of this interface or of any other interfaces it extends from.

The solution that forges a single class implementing both interfaces, in favour of two separated classes implementing both interfaces, is shown below:

```java
class MyCardDealer implements CardDealer {
    void draw() {/*code to flip top card*/}
}
class MyGraphicalComponent implements GraphicalComponent {
    void draw() {/*code to render on default device*/}
```
This solution, however, has a direct implication on the coupling and the cohesion of the new classes.

Symptom
The symptom of name clash errors is exhibited in terms of the weak cohesion of methods of classes that implement interfaces with identical method names.

Trigger
Errors associated with method clashes in interfaces can be exposed by review or inspection, which can help in identifying the methods that are inherited by more than one interface and that share the same signature. Sometimes compilers can detect these errors. If such methods share the same signature, but not the return type, a compiler error will be the symptom of the clash in method names. For example, if an interface CardDealer were to be specified exactly as shown earlier, and if the interface GraphicalComponent were to contain method int draw() (instead of the original void draw()), and the graphical CardDealer class were to attempt the implementation of void draw(), a compiler error will be generated.

Additions in the Revised Version of the Catalogue of Code Errors
Examples: None.

References
3.6 Object Initialisation

Original Version of the Catalogue of Code Errors

Error Name: Object Initialisation Error
Java has built in mechanisms that help ensure the proper initialisation of the instance variables of a newly created object. Such mechanisms ensure that if instance variables are not explicitly initialised by the developers, Java will automatically award them predictable default initial values which depend on the type of instance variable. For example, if an instance of type MyObject has an instance variable x of type int which is not initialised by the developer, value 0 will automatically be assigned to instance variable x. While this automatic instance variable initialisation reduces the likelihood of initialisation-related errors, it does not totally eliminate them. One cannot rely on default initialisation, rather one should attempt to explicitly and properly initialise all instance variables of an object during its construction. The following example shows that reliance on an improperly initialised object may cause errors.

```java
class MyDate {
    int month, day, year;

    public MyDate() {} //No argument constructor

    public MyDate() { /*code to return the next Day*/}

    public void toString() { /*code to print contents*/}

    public void nextDay() { /*code to return the next Day*/}
}

class TestMyDate {
    public static void main(String args[]) {
        MyDate testDate = new MyDate();
        System.out.println(testDate); //will print date: 0/0/0
        testDate.nextDay();
        System.out.println(testDate); //will print date: 0/1/0
    }
}
```

Output:
0/0/0 //meaningless date object
0/1/0 //method nextDay() worked on a meaningless date object to modify it into meaningless date object

As illustrated in the example, the state of the testDate object is initialised to an erroneous date and any calls made to the nextDay() method keep producing incorrect results.

Severity
Medium Error

Origin
Code Phase. Object Initialisation.

Causes
Mainly, the cause for incorrect object initialisation relates to the fact the developers have failed to understand the requirements of an object.
Prevention

The following is a set of guidelines to help avoid possible problems associated with object initialisation which are caused by object initialisation errors. The set of guidelines is explained below:

i) An object's instance variables should be declared private in order to avoid direct access from other objects. Accessor and mutator methods (e.g. getInstanceVariable() and setInstanceVariable()) should be the only way that an object's instance variables can be accessed and modified by other objects. For example, class MyDate shown earlier, could be equipped with methods such as void setMonth(int month) or int getMonth() to control access to the instance variable month.

ii) The failure of developers to fully understand how Java's mechanism for object initialisation works could be another cause of object initialisation errors. This would require developers to raise their awareness of all initialisation mechanisms.

iii) Sometimes the cause of incorrect object initialisation relates to the fact that the developers have failed to understand the requirements of an object. For example, a developer coding a TrafficLight class may have failed to understand that any object of type TrafficLight needs to have three instance variables initialised to three constants representing the three colours of a traffic light: green, amber, red. Such values are referred to as the "default natural values" of an object. While not every object can have default natural values, Java developers should initialise object instance variables to default values if such values exist. Otherwise, they should use constructors to initialise instance variables to known valid initial values.

iv) This is a corollary guideline to guideline number iii). Constructors should be equipped with code which checks the validity of the data used to initialise object instance variables. Such code should be compliant with the specification, and exceptions should be automatically thrown when invalid data are detected.

The above guidelines are not theoretical; their objective is to help developers to acquire a mindset conducive to good design to avoid errors related to object initialisation.

Symptom

Because the MyDate class example shown earlier suggests that if objects are not properly initialised, it is possible for them to start their life with an incorrect state, which results from incorrect instance variable values. As objects with incorrect or invalid states are used, they are likely to yield incorrect or unpredictable behaviour.

Trigger

The invalid object states can be exposed by allowing developers to bypass Java's encapsulation mechanism using getInstanceVariable() and setInstanceVariable() to report and modify the values of the instance variables, respectively.

Additions in the Revised Version of the Catalogue of Errors

Examples: None.
References


3.7 Accessing Methods versus Accessing Instance Variables

Original Version of the Catalogue of Code Errors

Error Name: Accessing Methods vs Accessing Instance Variables

In Java the type of reference and the type object the reference is referring to play an important role in determining the method or instance variable that can be accessed. In general, a Java object does not necessarily have to be referred to by a reference of the same type as the object. For example, inheritance allows a superclass type of reference to refer to a subclass type of object. If a method is invoked via an object reference, the actual type (i.e. the class the object is instantiated from) of the object determines the implementation of the method that is to be used. This rule is intuitive.

When instance variables are involved, however, the rule is different and counter intuitive. When an instance variable is accessed, the declared type of the reference is used, rather than the actual type of object that the reference is referring to, as one would expect. The example shown below illustrates the difference between the rules of accessing methods and accessing instance variables.

class Supershow {
    public String str = "superStr";
    public void show() {System.out.println("Super.show: " + str);
    }
}

class ExtendShow extends Supershow {
    public String str = "ExtendStr";
    public void show() {System.out.println("ExtendShow.show: " + str);
    }
}

public static void main(String args[]) {
    ExtendShow ext = new ExtendShow();
    Supershow sup = ext;
    sup.show();
    ext.show();
    System.out.println("Sup.str = " + sup.str);
    System.out.println("Ext.str = " + ext.str);
}

Output:

Extend.Show: ExtendStr
Extend.Show: ExtendStr
Sup.str = SuperStr
Ext.str = ExtendStr

The example features two classes related by inheritance. The superclass is called Supershow and contains an String instance variable (i.e. str), whose content (i.e. SuperStr), is accessed and displayed by method show(). The subclass, is called ExtendShow, which overrides method show() to access and display the content of instance variable str. The str instance variable has been redefined in ExtendShow to a new string value, namely, "ExtendStr".

The example also shows the definition of the sup superclass (i.e.Supershow) reference, and ext subclass (i.e.Extendshow) reference. The sup reference is pointing to an
Extendshow type object. The exact method to be called is determined at run-time, resulting in the invocation of the extendshow version of show(). When sup, which is referencing an extendshow type object, accesses the str instance variable, the superclass SuperStr rather than the extendshow ExtendStr instance variable is accessed. This situation is likely to cause errors.

Severity
Medium Error

Origin

Causes
A lack of awareness of the rules discussed above may cause errors to be introduced in situations similar to the example shown earlier.

Prevention
Establishing awareness of these rules, (especially in the case when instance variable access via superclass and subclass references is concerned) is the first step towards avoiding errors. Another way to avoid errors associated with the above access rules is to always avoid direct access to instance variables. For instance, in the example shown earlier, the instance variables are accessed directly:

```java
public void show() {System.out.println("Super.show: " + str);}
```

Instance variable access should be carried out by using specialised methods, such as accessor and mutator methods (e.g. getInstanceVariable() and setInstanceVariable()).

Symptom
These errors exhibit themselves when the wrong value is assigned to the wrong instance variable, resulting in subclass and/or subclass objects with possible incorrect or invalid object states.

Trigger
Errors associated with rules with regard to accessing methods and instance variables can be detected with code reviews and inspections.

Additions in the Revised Version of the Catalogue of Code Errors

Examples: None.

References


3.8 Cast Down the Inheritance Hierarchy

Original Version of the Catalogue of Code Errors

Error Name: Cast Down the Inheritance Hierarchy
Java allows one type of object reference to be assigned another type of reference. Such assignments, however, are governed by certain rules. When the assignment rules are violated, assignment-related errors occur. An assignment between different types of references is possible if such references refer to objects which have been instantiated from classes that belong to the same inheritance hierarchy.

The process of assigning the reference of one type to the reference of another type as type changing. Type changing may be performed automatically by the compiler in which case it is called implicit conversion or upcasting. The implicit conversion rules can be generalised by saying that in general, object reference conversion is allowed when the direction of the conversion is “up” the inheritance hierarchy. This means that in Java, it is acceptable to assign a subclass reference to a superclass reference (i.e. converting or changing the type of a subclass reference to a superclass). The code shown in the following example illustrates implicit conversion.

class Citrus {
    /*code to implement class Citrus*/
}

class Tangelo extends Citrus {
    /*code to implement class Tangelo*/
}

class Test {
    Tangelo tangelo = new Tangelo();
    Citrus citrus = tangelo;        //implicit conversion
    Citrus citrus2 = new Citrus();
    Tangelo tangelo2 = citrus2;     //Error

    Tangelo tangelo2 = (Tangelo) citrus2;
}

The example shows a Tangelo being converted into a Citrus (Citrus citrus = tangelo;). The system allows such conversion, because its direction is “up” the inheritance hierarchy. Essentially, this is allowed because of the “is a” relationship between Tangelo and Citrus established by inheritance.

However, if conversion “down” the inheritance hierarchy is attempted (e.g. in class Test: Tangelo tangelo2 = citrus2;) the compiler will reject it and flag an error. This is because now the relationship Citrus is Tangelo (?!?!) implied by the conversion, Tangelo tangelo2 = citrus2; is untrue. In this case, a Java developer can explicitly force the conversion to take place. Conversion or type changing that is explicitly performed by the Java developer for an assignment that would otherwise be rejected by the Java compiler is called casting or downcasting and can be accomplished using the cast operator. In the above example, the line which generated the error can be modified to:

Tangelo tangelo2 = (Tangelo) citrus2;
The newly inserted code: (Tangelo), is called a cast operator and its inclusion means that the Java developer is aware that the conversion is invalid for the compiler, but he/she still wishes to go ahead with it.

While casting is common in practice, it can be dangerous. For example, the following statement, where the assignment is carried out, survives compilation:

\[ Tangelo\ tangelo2 = (Tangelo)\ citrus2; \]

Nevertheless, an error will be generated and flagged where the statement is located when attempts are made to run the application. This occurs because the tangelo2 reference is still referring to a citrus type object (i.e. the same object that the citrus2 reference is referring to) prior to the assignment.

Severity
 Medium Error

Origin
 Code Phase. Data Handling.

Causes
In the literature, it is implied that cast down inheritance hierarchy errors may occur because Java developers do not fully understand conversion and casting rules or neglect to apply them properly.

Prevention & Trigger
In order to avoid or expose problems associated with casting, the exact type of the object which a reference is pointing to, should be verified before the object is allowed to get involved in any operation. Object verification can be accomplished using the Java instanceof operator. For instance, checks like the following can be added to code:

\[ \text{if (tangelo2 instanceof Tangelo)} \]

Such checks will improve Java developers' chances of avoiding errors related to casting.

Symptom
With errors that are associated with casting down the inheritance hierarchy, exceptions called (java.lang.ClassCastException) are thrown.

Additions in the Revised Version of the Catalogue of Code Errors

Example: Cast Down the Inheritance Hierarchy
class Mammal {
    /*code for Mammal class*/
};

class whale extends Mammal {
    /*code for Whale class*/
};

class Test {
    public static void main(String args[]) {
        Mammal m = new Mammal();
        Whale w;
        w = (Whale)m; //explicit class. compiler accepts it.
    }
}

java.lang.ClassCastException will be thrown at run-time.

References

3.9 Confusing == with equals()

Original Version of the Catalogue of Code Errors

Error Name: Confusing == with equals()
In Java, confusion often arises on the issue of equality. Such confusion exists with respect to the method equals() and operator ==. The method equals() performs a character-by-character comparison on two objects of type String and it can only be used with object references (e.g. String objects) and not primitive data type variables (e.g. int, double, etc.). Operator == tests if two references are referring to the same object. Method equals() and operator == are, therefore, different and may produce different results. The following example illustrates such difference.

class stringExample {
    public static void main(string args[]) {
        String s0 = "Programming";
        String s1 = new String("Programming");
        String s2 = s0;
        System.out.println("s0.equals(s1): " + (s0.equals(s1)));
        System.out.println("s0.equals(s2): " + (s0.equals(s2)));
        System.out.println("s0 == s1: " + (s0 == s1));
        System.out.println("s0 == s2: " + (s0 == s2));
    }
}

Output:
s0.equals(s1): true
s0.equals(s2): true
s0 == s1: false
s0 == s2: true

In the above example, the String type references s0 and s1 refer to two different String type objects, whose content is identical: Programming. The String type reference s2 refers to the same String type object as s0. This explains the different results obtained in the output section:

...s0.equals(s2): true
s0 == s1: false
...

Severity
Medium Error

Origin
Code Phase: Data Handling

Causes & Prevention
Lack of awareness of the difference between method equals() and operator == may cause Java developers to commit these errors.

Trigger
Such errors can be exposed by duplicating object comparisons using the == operator and equals() method and observing the results.
Symptom
The occurrence of errors associated with confusion between == and equals() can lead to incorrect software results or unpredictable object behaviour.

Additions in the Revised Version of the Catalogue of Code Errors

Example: Confusing == with equals()

class Test {
    public static void main(String args[]) {
        Integer ia = new Integer(10);
        Integer ib = new Integer(10);
        SYSTEM.OUT.PRINTLN("IA==IB IS " + (IA==IB));
        //prints "is false" because ia and ib are two different
        //objects
        //error hazard
        system.out.println("ia.equals(ib) " + (ia.equals(ib)));
        //prints "is true"
    }
}

References

3.10 Issues with Inner Classes

Original Version of the Catalogue of Code Errors

Error Name: Issues with Inner Classes

An inner class as a class which is defined in another class. The class where the inner class is defined is called the enclosing class (also known as the nesting or outer class). Inner classes are used mostly to implement event handling. There are two rules with respect to the access rights of inner classes, which if ignored can lead to errors. The first rule states that an inner class can access and modify all members (i.e. instance variables and methods) of the enclosing class. This can be achieved by accessing or modifying the required member directly with the need to qualify its name with the name of the enclosing class. The second rule is not commonly stated in Java references. This rule states that while an object of the inner class is always associated with an object of the outer class, the opposite is not true. An object of the outer class does not necessarily have to have any inner class objects associated with it. The following example illustrates a situation whereby the failure to respect the above rules can constitute an error hazard.

```java
class OuterOne {
    private int x;
    public int getX() {return x;}
    public void outerMethod() {System.out.println("x is: "+x);}

    public class Innerone {
        int y;
        public void innerMethod() {++x; //error hazard
            System.out.println("enclosing x is: "+x);
            System.out.println("y is: "+y);
        }
    }
}
class OuterOneTest {
    public static void main(String args[]){
        OuterOne.Innerone innerone = outerOne.new Innerone();
        innerone.innerMethod();
        OuterOne.Innerone innerTwo = outerOne.new Innerone();
        innerTwo.innerMethod();
    }
}
```

Output:
enclosing x is : 1
y is 0
enclosing x is : 2
y is 0
x = 3

This example features a class outerone, which contains and inner class called innerone. Class InnerOne contains a method called innerMethod(), which accesses and modifies the instance variable x of class outerone. Class OuterOneTest instantiates multiple instances of the class innerone. Clearly, the example is an embodiment of the two rules stated above. It also shows that multiple instances of the inner class are constructed (e.g. innerone, innerTwo, etc.). Such instances have access to the enclosing class instance variables. If the code in the inner class (e.g. public void innerMethod()) is not carefully designed, it has the potential to change the
enclosing class instance variables to values (e.g. int x) which may not comply with the specification.

Severity
Medium Error

Origin

Causes & Prevention
Lack of awareness of such rules can be the cause of errors when dealing with inner classes.

Trigger
In order to be able to expose any errors or error hazards that are associated with inner classes, developers need to getEnclosingClassInstanceVariable() type methods in order to be able to access and monitor the values of enclosing class instance variables continuously during the object's lifetime.

Symptom
Errors associated with inner classes can result in incorrect or invalid states of instances of the enclosing class and can lead to the unpredictable behaviour of objects.

Additions in the Revised Version of the Catalogue of Code Errors
Examples: None.

References
3.11 Memory Leaks

Original Version of the Catalogue of Code Errors

Error Name: Memory Leaks
In Java, like in other languages, objects occupy memory. Such memory needs to be reclaimed and returned to the operating system when an object has served its purpose and is no longer needed. If memory is not reclaimed and returned to the operating system, the chances are that eventually there will be insufficient memory for the application to run, exhibiting poor software performance, software crashes or java.lang.OutOfMemoryErrors. These situations are commonly known as memory leaks. While in other languages (e.g. C++) developers are responsible for writing code that allocates memory to objects and reclaims it, in Java the task of reclaiming memory is performed automatically. The utility program that performs such task is called a garbage collector. A garbage collector runs automatically in the background constantly searching for objects that are unreachable (i.e. no longer needed) by any other object in the software. This has led to the misconceived popular belief that memory leaks are not possible in Java programs. Unfortunately this is not always the case. The following example illustrates the possibility of memory leaks in a Java program.

class LeakExample {
    public static void main(string args[]) {
        int big_array[] = new int[100000];

        // this queries big_array and obtains a result
        int result = compute(big_array);

        // At this point big_array is no longer needed. However,
        // it will be garbage collected only when the method
        // returns, because big_array is a local variable.

        // The method, however, will never return, because, it
        // loops infinitely, handling the user's input.
        for (;;) {handle_input(result);}
    }
}

The example shows the invocation of method compute(big_array), which queries big_array to obtain a result. After the result is obtained (i.e. after statement int result = compute(big_array); executes) big_array is no longer needed. Yet, big_array will still remain in memory and can only be garbage collected when the method (i.e. method main(string args[])) returns. This occurs because big_array is a local variable. Method main(string args[]), however, will never return because it is supposed to loop infinitely in order to handle the user's input. This is a situation where an object (big_array) is left unused in memory. Situations like this are likely to cause memory leaks and cannot be resolved by the Java garbage collector.

Severity
Medium Error
Origin
Code Phase, Data Handling.

Causes & Prevention
Memory leaks can be caused by many factors:

i) Firstly, unwanted object references can cause memory leaks. The easiest way to avoid these memory leaks is to identify objects that are no longer needed and assign their references to null. For instance, in the example presented earlier (class LeakExample), the statement: big_array = null; could be inserted after the result variable is initialised (i.e. after statement int result = compute(big_array); executes). The act of assigning null to a variable marks the object as able to be referenced by the variable for garbage collection.

ii) A second cause for memory leaks is the developers' failure to free native system resources. Native system resources are typically allocated through the Java Native Interface (JNI) by functions that are external to Java implemented in C or C++. For example, Java developers commonly use Abstract Windowing Toolkit (AWT) classes (e.g. Frame, Graphics, etc.) and when such classes are no longer needed, they fail to release the system resources reserved for them using the dispose() or finalize() methods.

iii) A third cause of memory leaks is when Java developers reuse third-party libraries (e.g. Java Development Kit (JDK), etc.) or just code developed by other developers. The reused code may already have errors due to the two causes discussed above. One way to avoid existing memory leaks in libraries is to become acquainted with them by checking errors that are published by the developers of such libraries. For example, Sun's Java Developer Connection Bug Database publishes common errors in JDK libraries.

iv) Developers should carefully program Java software that includes structures that are likely to cause memory leaks. These include collection classes, such as hashable, vectors, arrays, etc. In such cases, developers need to study the lifetime of objects of such classes and assign their references to null when such objects are no longer needed. A Java programmer may even explicitly call the garbage collector to execute using method System.gc(). However, calls to System.gc() do not guarantee the prompt execution of the garbage collector and the order in which any objects marked for garbage collection will be cleared. The call to System.gc() is just a suggestion to the Java Virtual Machine for the garbage collector to be called. This suggestion can be ignored.

v) Another structure that commonly introduces memory leaks includes event listeners. Event listeners may cause memory leaks when an object is added to an event listener list, but not removed, when the object's usefulness has lapsed. Memory leaks involving lapsed listeners can be avoided by always pairing calls to addEventListener() and removeEventListener().

vi) Finally, objects that are accessed from static structures are also prone to memory leaks. Static structures include instance variables, methods or even classes, which, once initialised, will stay in memory for as long as the program that defines them stays in memory. This implies that any object that is referred to by a static structure will be kept in memory, although such an object may not actually be needed, thus, increasing the possibility for a memory leak.
Trigger
In order to detect memory leak errors, memory profiles of Java software that are suspected to result in memory leaks should be developed. A memory profile portrays the memory requirements of the software at various points during its execution. Automatic tools such as Rational Purity which creates memory consumption profiles can be used for this purpose. Other tools include JProbe, Optimize!, JInsight, etc. Such tasks, however, can also be performed by either writing specialised classes or by using operating system tools which help observe the memory needs of an application or process or by using Java Development Kit methods. For example, in class java.lang.Runnable, methods freeMemory() and totalMemory() return the amount of unused and total memory, respectively.

A memory profile will help developers localise the memory “hotspots” and, thereby, make informed decisions on further steps to optimise memory consumption. Memory profiles may not only help prevent memory leak errors but may also expose their presence, if such errors have already been committed. Although Java, with the introduction of the automatic garbage collector has reduced the potential for memory leaks, they have not been eliminated. Misconception about the capabilities of the garbage collector could itself be a cause for memory leaks. Memory leaks can be avoided when awareness about them and the structures that are prone to them is established.

Symptom
If memory leaks occur in a Java application, the chances are that eventually there will be insufficient memory for the application to run, exhibiting poor software performance, software crashes or java.lang.OutOfMemoryErrors.

Additions in the Revised Version of the Catalogue of Code Errors
Examples: None.

References
3.12 Thread Deadlocks

Original Version of the Catalogue of Code Errors

Error Name: Thread Deadlocks

Threads are the tool that the Java language provides to achieve the execution of multiple tasks concurrently. A thread is commonly defined as a single sequential flow of control in a program. This "single sequential flow of control" may be responsible for a task that a program is expected to perform (e.g. see figure 1 (a)). The notion of a single thread, however, does not add any value to what is currently known about programs, because every program is a single sequential flow of control. The real value is added when multiple threads are used in a program to perform many tasks simultaneously (see figure 1 (b)). For example, within a browser, a user may scroll a page, download a file and play a sound concurrently.

![Figure 1 - Threads](image)

Sometimes multiple threads may need and attempt to access the same object concurrently. This situation can be illustrated with an inventory e-commerce software example. Assuming that there are two threads (representing two independent customers) attempting to obtain the same last item in an inventory object simultaneously, both threads would have determined the availability of the item. However, it is possible that one thread would charge the customer and actually obtain the item, while the other thread would charge the customer and realise that the item has actually been sold out. The possibility of the occurrence of such a scenario requires the object (i.e. the inventory object) to indicate to the contending threads (i.e. the customer objects) whether it is being used by another thread. Such indication would prompt the contending threads to wait until the object (i.e. the inventory object) is free again (see figure 1 (c)). The mechanism that helps accomplish this is known as the Java monitor model and the object that different threads contend for is called a lock object. While the example in figure 1 (c) shows two threads are contending for the same object, in
practice, two or more threads may be contending for two or more objects. When the number of objects that are being contended for by the threads is more than one, thread deadlocks are possible. A deadlock occurs when two or more threads indefinitely wait for the other(s) to do something. The scenario in figure 2 illustrates a thread deadlock situation.

![Thread Deadlocks Diagram](image)

Figure 2 – Thread Deadlocks (Mitchell, 2000)

Figure 2 shows that Thread A will not release its hold on object X until it obtains Object Y. Likewise, Thread B will not release its hold on object Y until it obtains Object X. This scenario suggests the Threads A and B will be waiting for each other indefinitely, resulting in a deadlock.

Severity
Medium Error

Origin
Code Phase. Data Handling.

Causes
When threads that attempt to use more than one object are not properly designed, they are likely to cause thread deadlocks. Thread deadlocks may also be caused when Java developers possess inadequate knowledge about threads in general or about possible deadlock scenarios in particular.

Prevention
It is the programmer's responsibility to prevent thread deadlocks from occurring. The technique that is commonly used to avoid thread deadlocks is called resource ordering. With this technique, an ordering value is assigned to all objects whose locks must be
acquired by a set of contending threads. It is the programmer's task to ensure that the locks on all objects are always acquired and released in that order. This ensures that:

"It is impossible for two threads to hold one lock each and be trying to acquire the lock held by the other—they must both request the locks in the same order so once one thread has the first lock, the second thread will block trying to acquire that lock, and then the first thread can safely acquire the second lock." (Arnold, Gosling, & Holmes, 2000, p. 254).

Trigger

Thread deadlocks can be exposed by running the application which is suspected to contain thread deadlock errors many times. For instance, in figure 2, it is possible for Thread A to lock object X and object Y before Thread B even starts its execution, in which case the deadlock will not occur. Alternatively, it is also possible for Thread B, to lock object Y and object X before Thread A starts its execution, in which case the deadlock will again not occur. It is the responsibility of a utility program called the thread scheduler to determine which thread to start and when to start it. A Java developer does not have control over the thread scheduler. It is, therefore, important that any program suspected of ending up in a deadlock be run many times to trigger the presence of any deadlocks.

Symptom

The symptoms of a thread deadlock occur when the application is 'stalled' or 'freezes' or 'hangs'.

Additions in the Revised Version of the Catalogue of Code Errors

This section contains two examples: a) Thread Deadlock, and b) Thread Deadlock resolved with RESOURCE ORDERING

Example: Thread Deadlock

class Friendly {
    private Friendly partner;
    private String name;

    public Friendly(String name) {
        this.name = name;
    }

    public synchronized void hug() {
        System.out.println(Thread.currentThread().getName() + " in " + name + ".hug() trying to invoke "+ partner.name + ".hugBack()";
    }

    private synchronized void hugBack() {
        System.out.println(Thread.currentThread().getName() + " in " + name + ".hugBack()");
    }

    public void becomeFriend(Friendly partner) {
        this.partner = partner;
    }

    public static void main(String args[]) {
        final Friendly robert = new Friendly("Robert");
    }
}
```java
final Friendly roberta = new Friendly("Roberta");
roberta.becomeFriend(roberta);
new Thread(new Runnable() { public void run() { roberta.hug(); } }, "Thread1").start();
//new thread started
new Thread(new Runnable() { public void run() { roberta.becomeFriend(roberta); } }, "Thread2").start();

This will result in an output similar to the following, after which the program will hang its execution.

Thread1 in roberta.hug() trying to invoke roberta.hugback()
Thread2 in roberta.hug() trying to invoke robert.hugback()

Example: Thread Deadlock resolved with Resource Ordering

class Friendly1 {
    private Friendly1 partner;
    private String name;
    private int order = 0;
    private static int i = 0;
    Friendly1 first, second;

    public Friendly1(String name) {
        this.name = name;
        i = i + 1;
        order = i;
        //creating objects with a unique identifier, namely order
    }

    public void involveFirstAndSecond() {
        first = this;
        System.out.println(first.name);
        second = partner;
        System.out.println(second.name);
    }

    public synchronized void hug() {
        involveFirstAndSecond();
        //ensure locks are obtained in the specified order
        while (first.lockorder() < second.lockorder()) {
            try { wait(); } catch (InterruptedException e) { }
        } //end while

        System.out.println(Thread.currentThread().getName() + " in " +
                           this.name + ".hug() trying to invoke " + partner.name +
                           "hugback()" + " order=" +
                           this.lockorder());
        partner.hugback();
    }

    public synchronized void hugBack() {
        System.out.println(Thread.currentThread().getName() + " in " +
                           this.name + ".hugBack()" + " order=" +
                           this.lockorder());
        notify();
    } //notify threads who need the lock on higher order objects

    public void becomeFriend(Friendly1 partner) {
        this.partner = partner;
    }
}
```
public int lockOrder() {
    return order;
}

public void setOrder(int o) {
    order = o;
}

public static void main(String args[]) {
    final Friendly1 robert = new Friendly1("Robert");
    final Friendly1 roberta = new Friendly1("Roberta");
    robert.becomeFriend(roberta);
    roberta.becomeFriend(robert);
    new Thread(
        new Runnable() {
            public void run(){
                robert.hug();
            }, "thread1").start();

    new Thread(
        new Runnable() {
            public void run(){
                roberta.hug();
            }, "Thread2".start();

This example will result in the output similar to the following:

Thread1 in robert.hug() trying to invoke roberta.hugback()
Thread1 in roberta.hugback()
Thread2 in roberta.hug() trying to invoke robert.hugback()
Thread2 in robert.hugback()

References


Additional Errors for Future Upgrades of Catalogue of Errors

Other Minor Errors but Equally Dangerous Errors

1. == vs ===
The programmer may have substituted = for ==. If a and b are both of type boolean, the java compiler will not be able to flag if (a=b) {} as an error. It will perform a perfectly legal assignment instead.

class Test1 {
    public static void main(String args[]) {
        boolean a, b;
        a=b=true;
        if(a==b) {//comparison operator
            System.out.println("In first if statement");
            //will print out
        }
        if(a=b) {//assignment operator
            System.out.println("In second if statement");
            //will print out
        }
    }
}

2. Misplacing the Semicolon (;) after Comparison in an if Statement
If a semicolon is misplaced after the comparison in an if statement, it is possible that an unintended error has just been introduced. For instance, in the example shown below, the original intention is for line:

    System.out.println("This line will always print");

    to print, if and only if a==b. However, if a semicolon is unintentionally inserted right after the comparison, the above line will always be executed. The Java compiler will not flag this situation as an error.

class Test2 {
    public static void main(String args[]) {
        int a, b; a = 1; b = 2;
        if (a==b); //misplaced semicolon
        System.out.println("This line will always print");
    }
}
3. Missing Braces Around a While Loop Body

It is possible that a programmer might forget to insert braces to delimit a loop body. Failure to do so may lead to incorrect results. For instance the following code when run will generate conflicting results.

```java
class Test3 {
    public static void main(String args[]) {
        int a, b, x, y, w, v; a = 1; b = 2; x = y = w = v = 0;
        while (a==b) {
            x++;
            y++;
            System.out.println("x=", x + " and y=", y);
            //will print x=1 and y=1
        }
        while (a==b) {
            w++;
            v++;
        } //end while
        System.out.println("w=", w + " and v=", v);
        //will print w=1 and v=1
    }
}
```

A similar error may happen if braces are omitted for if and else branches whose bodies have more than one statement.
Appendix B: INSTRUMENTATION

Table of Content

2.1 DEVELOPMENT INSTRUMENTS .............................................................................................................. 476
  2.1.1 Problem Statement: Intersection Simulator ..................................................................................... 476
  2.1.2 Problem Statement: Hotel Lobby Simulator ...................................................................................... 480
  2.1.3 Similarities between Intersection and Hotel Lobby Simulator ....................................................... 484
  2.1.4 Development Artifact Templates .................................................................................................... 485
2.2 DATA COLLECTION INSTRUMENTS ..................................................................................................... 488
  2.2.1 Participant Instruments .................................................................................................................... 488
    2.2.1.1 Software Development Background Questionnaire ....................................................................... 488
    2.2.1.2 Error Framework Evaluation Questionnaire ................................................................................ 490
    2.2.1.3 Catalogue of Errors Evaluation Questionnaires .......................................................................... 491
    2.2.2 Review instruments ......................................................................................................................... 494
      2.2.2.1 Artifact Error Report Forms ...................................................................................................... 494
      2.2.2.2 Artifact Review Guides ............................................................................................................. 497
2.3 TRAINING INSTRUMENTS ..................................................................................................................... 502
  2.3.1 Error Framework and Catalogue of Errors Training Instruments .................................................. 502
2.4 ETHICS INSTRUMENTS ........................................................................................................................ 533
  2.4.1 Statement of Disclosure .................................................................................................................... 533
  2.4.2 Consent Form ................................................................................................................................... 533
2.1 Development Instruments

2.1.1 Problem Statement: Intersection Simulator

EDITH COWAN UNIVERSITY

CSP3241

Internet and Java Programming

Intersection Simulator

Author: Dr JWL Millar
Thursday, 16 February 2001
Problem statement

The intersection of Hay and Williams Streets in central Perth is being revamped. The traffic in William St is one-way travelling south. Hay St is open to traffic west of William St and that traffic flows one-way namely east. Hay St east of William St is a pedestrian zone known as Hay St Mall and no traffic is permitted to enter from this intersection. Consequently the east-bound traffic in western Hay St must turn right at William St and travel in the only direction allowed in William St namely south. In addition to vehicular traffic, there are pedestrians who wish to cross the intersection.

The Town Council has decided to install traffic lights to control the traffic and pedestrian movements. These lights will be controlled by sensors in the roads, indicating a car waiting to cross, and by push buttons operated by pedestrians. The road sensors are behind the stop-lanes on the northern side of the intersection across all lanes in William St and on the western side of the intersection across all lanes of Hay St. The pushbuttons are mounted on the traffic light poles in four locations. On the north-western corner of the intersection is one button for pedestrians wishing to cross either William St or Hay St. A similar button is located on the south-western corner of the intersection. Two further buttons are mounted on the eastern side of the intersection for people wishing to cross to either side of the western section of Hay St.

When a car approaches a red light it will stop and the sensor will indicate that the cycle of traffic light changes will need to advance to eventually allow the car to pass through the intersection. A car approaching a green light will continue through the intersection. When a pedestrian push button is pushed, the cycle of traffic light changes will need to advance to eventually halt all vehicular flow and indicate that pedestrian traffic can cross the intersection. The push buttons also emit one of two sounds one of which indicates no movement whilst the other indicates that crossing of the intersection is allowed.

In order to study the proposed traffic light system, the council has decided to commission the students in this unit to prepare a simulation of the intersection on a computer screen. At this stage, some simplifying assumptions have been specified such as assuming that both streets only carry one lane of traffic. Furthermore, the cycle of the traffic lights is simplified and set out below. A maximum of one car will be visible in each street at any time.

The simulation will display a representation of the intersection including the vehicles and pedestrians as well as the roads, traffic lights and surroundings (not necessarily true to life). The simulation display will include two buttons, a clock-like display and a simulation of a push button:

- The first button will cause a vehicle to approach the intersection along Hay St and the other button will cause a pedestrian to approach the intersection. Cars travelling along William St will appear at random intervals.

- A clock-like display will indicate state of the traffic lights as they move through the cycle of traffic light patterns. The traffic lights will stay in any given state for at least 5 seconds. After the 5 seconds and once one or more buttons have been pressed or sensors activated, the traffic lights will move to the next state. The sequence of states, if all are requested, will be:

  1. Pedestrians can walk and then
  2. Cars in William St can proceed and then
  3. Cars in Hay St can proceed and then back to 1

- A display representing the push buttons on the traffic lights will include the button, which will flash once when any button has been pressed, and a "Wait" light that will illuminate and remain illuminated until pedestrians are permitted to cross the intersection.
Figure 1. The basic GUI for the simulation
REQUIRED FUNCTIONALITY

User Documentation

Instructions

Functionality

Basic Graphical Elements

- Intersection drawn
- 2 Buttons to create people/Hay St cars drawn
- Traffic lights drawn
- Pushbutton panel drawn
- Surroundings drawn
- Clock drawn

Basic Functional Elements

- push Pedestrian Button - person appears and presses button on pedestrian phase, person crosses intersection
- push vehicle button - car approaches along Hay St.
on Hay St phase, car waiting crosses intersection
- clicking sounds for pedestrian button press
- William St cars appear randomly
- Road sensors work
- Push button panel works

Advanced Features

- Allows for queues to form
- Traffic lights and clock synchronised - finer animation
- Priorities are correct

Extras

- ...
- ...
- ...
- ...
2.1.2 Problem Statement: Hotel Lobby Simulator

EDITH COWAN UNIVERSITY

CSP3241
Internet and Java Programming

Hotel Lobby Simulator

Author: Dr JWL Millar
Thursday, 12th July 2001
Problem statement

The Setting:

The Excelsior Hotel has a reception counter in the lobby that is always staffed by at least one staff member. The duties of the staff are to assist guests who come to the counter and to answer incoming phone calls.

Proposed Changes:

The Management of the Excelsior Hotel has decided to extend the services offered at the reception counter by adding a safe deposit room at one end of the counter. This room will have a staff entrance from behind the counter and a guest entrance from the hotel lobby. A guest wishing to use the safe deposit room must press a button outside the guest entrance in order to attract the attention of staff behind the counter. A staff member will respond to the button being pressed by entering the safe deposit room via the staff entrance and admitting the guest. Only one guest will be allowed in the safe deposit room at any time and staff assisting a guest in the safe deposit room must remain there until the guest has left the room.

The staff at the reception counter will now have an extra duty namely responding to a guest pressing the button outside the guest entrance to the safe deposit room. They will be advised that this duty is the lowest priority with the highest priority being to answer the phone.

In order to study the proposed changes, it has decided to commission the students in this unit to prepare a simulation:

The simulation will provide for guests to approach the counter at random intervals whilst allowing for phone calls to the staff to be initiated by person running the simulation. Similarly it will allow for guests wishing to use the safe deposit room to be generated by the person running the simulator.

The simulation will display the following representation (see figure 1 in section 6 in a later page):

- a plan of the lobby and counter area including the telephone and the safe deposit room;
- a button to initiate phone calls to the counter;
- a button to generate a guest approaching the guest entrance to the safe deposit room;
- a set of indicators used to indicate when the phone is ringing and when it is in use, when the button outside the guest entrance to the safe deposit room has been pressed and when that room is occupied;
- figures of guests and staff as they move around the lobby, counter area and safe deposit room;
- appropriate sounds as buttons are pressed and phones ring;
- lights on the doors to the safe deposit room to indicate when it is occupied;
- background music in the lobby.
Figure 1 – The basic GUI for the simulation
REQUIRED FUNCTIONALITY

User Documentation

Basic Graphical Elements
- Basic lobby area drawn
- Appropriate surroundings drawn
- Background music playing
- Figures for staff and guests drawn
- Indicator lights drawn
- 2 Buttons to create guests and phone calls drawn

Basic Functional Elements
- Pushing 'RingPhone' button – telephone indicator show phone ringing
- Pushing 'RingPhone' button causes phone to ring
- Pushing 'RingPhone' button causes staff to answer phone
- Safe deposit room door operates
- Pushing 'Safe Deposit Room' button causes guest to approach guest entrance
- Staff respond to guest requiring access to safe deposit room
- Indicator light shows status of safe deposit room
- Guests randomly approach reception counter

Advanced Features
- Allows for a second staff member behind reception counter
- Allows for queues to form
- Priorities for duties correct

Extras

..........................................................

..........................................................

..........................................................

..........................................................
2.1.3 Similarities between Intersection and Hotel Lobby Simulator

The objective of this document is to compare and contrast the Intersection and Hotel Lobby Simulators, the problem statements which were presented earlier. A set of criteria has been used for this purpose and have been summarised into three broad categories, namely:

i) Basic Graphical Elements,
ii) Basic Functional Elements,
iii) Advanced Features, and
iv) Extras.

The specific criteria that are included in each category shown above have been summarised in the table 1 below. Table 1 also indicates whether the two simulators shown earlier fulfil a given criterion.

<table>
<thead>
<tr>
<th>Criteria</th>
<th>Intersection Simulator</th>
<th>Hotel Lobby Simulator</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Basic Graphical Elements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Drawing the Interface</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2 Building two control buttons</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3 Drawing the simulator actors</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4 Building the indicators</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5 Drawing the surroundings</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Basic Functional Elements</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Random generation of actors</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2 User generation of actors</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3 Synchronisation and multi-threading</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>4 Inclusion of sound to indicate required action was carried out</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>5 Updating designated indicators to reflect program status</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>6 Animating the simulator actors</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td><strong>Advanced Features</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 Simulator actors with each other interact in accordance with designated priorities</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>2 Queues are allowed to be formed</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>3 Extras</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>
2.1.4 Development Artifact Templates

Internet and Java Programming (CSP3241)

REQUIREMENTS ARTIFACT TEMPLATE

I. Introduction
   A. Simulator Overview
   B. Software Project Constraints

II. Goal Definition and Description

III. Requirements Definition and Description
   A. Requirements Narrative
   B. Restrictions/Limitation

IV. Viewpoint Definition and Description

V. Scenario Definition and Description

VI. Validation Criteria

VII. Proof of Intellectual Property (Time Spent Log)

VIII. Appendices
     (e.g. Traceability Matrices, etc.)

N.B. This submission relates to the requirements artifact for the simulator.

Dr JWL Millar
5/3/01

Mr Indrit Troshani
I. Introduction

II. High Level Design
   A. Class Identification
      1. Attributes/Instance Variables
      2. Methods/Behaviours
   B. Inheritance Definition
   C. Object Collaborations/Interactions
   D. Human-Computer Interface

III. Low Level Design
   A. Algorithm Definition

IV. Validation Criteria

V. Proof of Intellectual Property (Time Spent Log)

Dr JWL Miller

5/3/01

Mr Indrit Troshani
CODE ARTIFACT TEMPLATE

1) Simulator Code
   The source code for your simulator application must be submitted and
   written according to the Code Conventions specified by Sun Microsystems
   (Refer to Lecture 9, Week 9). The following is an example.

   ```java
   import javax.swing.*;
   public class ExampleClass1 {
      private instanceVariableType instanceVariable1;
      private instanceVariableType instanceVariable2;
      public methodReturnType method1(parameterType p) {
         //statement1;
         //statement2;
      }
   }

   public class ExampleClass2 {
      private instanceVariableType instanceVariable1;
      private instanceVariableType instanceVariable2;
      public methodReturnType method1(parameterType p) {
         //statement1;
         //statement2;
      }
   }
   //etc...
   ```

2) Evidence of Testing

3) Proof of Intellectual Property (Time spent log)

   Dr JWL Millar
   Mr Indrit Troshani
   5/3/01
2.2 Data Collection Instruments

2.2.1 Participant Instruments

2.2.1.1 Software Development Background Questionnaire

SOFTWARE DEVELOPMENT BACKGROUND QUESTIONNAIRE

1. Student ID: ___________________________

2. Course: _________________________________

3. Do you have any previous software development experience? (Please, circle one!)
   Yes  No

4. If Yes, what is the nature of your experience? (Please, tick one or more!)
   □ Industry Experience  □ Classroom Experience  □ Other
   (Please, specify below!)

5. How do you rate your system development experience in the following paradigms
   (Please, tick one!)?
   
   Procedural Software Development
   
   Object-Oriented Software Development

6. In what aspect of the development lifecycle do you consider yourself proficient?
   (Please, tick one or more boxes!) and how do you rate your proficiency (Please, tick only one circle!)
   
   □ Requirements Specification
   
   □ High-Level Design
   
   □ Low-Level Design
   
   □ Coding
   
   □ Testing
7. Which of the following programming languages are you proficient at? (Please, tick one or more!)

☐ Ada ☐ C ☐ C++
☐ Visual Basic ☐ Oracle Programming ☐ Pascal
☐ Other (Please specify)

NOTE: THE INFORMATION COLLECTED IN THIS FORM WILL BE KEPT STRICTLY CONFIDENTIAL, DE-IDENTIFIED, AND USED SOLELY TO ASSESS PROGRAMMING BACKGROUND FACTORS AFFECTING RESEARCH RESULTS.
2.2.1.2 Error Framework Evaluation Questionnaire

Error Framework Evaluation Questionnaire

**Student ID:**

Please rate (✔) the Error Framework perspectives in terms of their usefulness, with 5 being useful and 1 being not useful.

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Useful (5)</th>
<th>Useful (4)</th>
<th>Useful (3)</th>
<th>Not useful (1)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Error Name</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Category/Origin</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Severity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Cause</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Prevention Guidelines</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Symptom</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Trigger</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

What other information do you think an error model should include? (use additional space if necessary?)

---

490
### Catalogue of Requirements Errors Evaluation Questionnaire

**Student ID:**

Please indicate (✓) how much you agree or disagree with the following statements!

<table>
<thead>
<tr>
<th>Statement</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalogue of Requirements Errors is easy to use.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors is easy to follow.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me understand requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me understand the origin of some requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped appreciate the severity of some requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me understand the cause of some requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me understand how to prevent some requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me understand the symptoms of some requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me identify triggers to expose some requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me identify some requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has helped me correct possible requirements errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Requirements Errors has improved my skills to specify higher quality requirements.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Catalogue of Requirements Errors can be improved as follows (use back if space insufficient):**

---

---

---

---

---

---

---

---

---
# Catalogue of Design Errors Evaluation Questionnaire

Student ID: __________________

Please indicate (✓) how much you agree or disagree with the following statements:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalogue of Design Errors is easy to use.</td>
<td>5</td>
<td>4</td>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Catalogue of Design Errors is easy to follow.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me understand design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me understand the origin of some design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped appreciate the severity of some design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me understand the cause of some design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me understand how to prevent some design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me understand the symptoms of some design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me identify triggers to expose some design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me identify some design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has helped me correct possible design errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Design Errors has improved my skills to specify higher quality design.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Catalogue of Design Errors can be improved as follows (use back if space insufficient):

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________

_________________________________________________________________________
Catalogue of Code Errors Evaluation Questionnaire

Student ID: __________________________

Please indicate (✓) how much you agree or disagree with the following statements:

<table>
<thead>
<tr>
<th></th>
<th>Strongly Disagree</th>
<th>Disagree</th>
<th>Neutral</th>
<th>Agree</th>
<th>Strongly Agree</th>
</tr>
</thead>
<tbody>
<tr>
<td>Catalogue of Code Errors is easy to use.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors is easy to follow.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me understand code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me understand the origin of some code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped appreciate the severity of some code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me understand the cause of some code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me understand how to prevent some code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me understand the symptoms of some code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me identify triggers to expose some code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me identify some code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has helped me correct possible code errors.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Catalogue of Code Errors has improved my skills to specify higher quality code.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Catalogue of Code Errors can be improved as follows (use back if space insufficient):

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________

________________________________________________________________________
2.2.2 Review Instruments

2.2.2.1 Artifact Error Report Forms

**Requirements Error Report Form**

Student ID: ________________

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Design Errors Identified in Design Artifact:</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1. Algorithmic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Interface</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3. Reuse</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Strong Coupling</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. Weak Cohesion</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6. Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Requirements Errors Identified in Design Artifact:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Omission</td>
</tr>
<tr>
<td>2. Ambiguity</td>
</tr>
<tr>
<td>3. Inconsistency</td>
</tr>
<tr>
<td>4. Incorrect fact</td>
</tr>
<tr>
<td>5. Other</td>
</tr>
</tbody>
</table>
Requirements Errors Identified in Code Artifact:

- 1. Ambiguity
- 2. Inference
- 3. Incorrectness
- 4. Incomplete
- 5. Other

Design Errors Identified in Code Artifact:

- 1. Abuse of Interfaces
- 2. Failure to establish invariants
- 3. Inconsistent Declaration
- 4. Constructor Inherited overidden...
- 5. Contracts from different interfaces
- 6. Object Initialization Error
- 7. Accessing Methods vs. Accessing
- 8. Class Down inheritance hierarchy
- 9. Comparing -- with equals
- 10. Issues with Inner Classes
- 11. Memory Leak
- 12. Thread Deadlocks
- 13. Other

Code Errors Identified in Code Artifact:

Student:

Code Error Report Form
2.2.2.2 Artifact Review Guides

1. Requirements Artifact Review Guide

1. Check the completeness of the requirements.

Incomplete requirements suggest the presence of omissions. An omission exists when requirements are not included or have not been specified in the requirements artifact.

2. Check the consistency of the requirements.

Inconsistent requirements suggest the presence of inconsistency errors. An inconsistency exists when two or more requirements that have been included in the requirements artifact conflict with each other.

3. Check the clarity of the requirements.

Lack of clarity in the requirement artifact suggests the presence of ambiguities. An ambiguity exists when a requirement has not been clearly represented in the requirements artifact. An ambiguity also exists when one can make multiple interpretations to the same term/concept/characteristic in a particular context.

4. Check the correctness of the requirements.

The presence of requirements that assert facts that are untrue under the conditions of the requirements artifact under review suggest the existence of incorrect fact errors.

Other Possible Requirements Errors

5. Check the relevance of the stated requirements.

The presence of information or requirements that are not needed or used signal the presence of extraneous information errors.

6. Check the conciseness of the stated requirements.

7. Check for good organisation of the requirements artifact

This Requirements Artifact Review Guide must be read in conjunction with the Catalogue of Requirements Errors, where more information about each error can be found.
2. Design Artifact Review Guide

1. Check for flaws in the correctness of algorithms.

The presence of flaws in algorithms indicates the presence of algorithmic errors. Flaws in algorithms include, effectiveness (e.g. algorithm does what it is supposed to do), decision/loop problems, clarity (e.g. indentation, nesting), efficiency (e.g. unnecessarily long/complex algorithms that perform simple tasks), failure to properly indicate algorithm inputs and outputs, preciseness of algorithm steps, executability (algorithm contains executable steps), etc.

2. Check the interfaces of the interacting objects.

Interacting objects can exhibit interface errors in two possible cases:

i) The client object does not call the server's services in a "legal" manner (i.e. does not provide the correct types, order, and number of the required input arguments); and,

ii) The client does not interpret the server's correct answers (i.e. does not assign the returned values to variables of the appropriate data type) properly.

3. Check the correctness of the inheritance hierarchies constructed by the developer for reuse errors.

In order to check for reuse errors check that:

i) No errors occur when subclass methods interact with superclass instance variables or methods; and

ii) No errors occur when superclass methods are reused in a subclass environment.

4. Check for unacceptable levels of coupling (i.e. strong coupling). Refer to following table:

<table>
<thead>
<tr>
<th>Interaction Coupling</th>
<th>Component Coupling</th>
<th>Inheritance Coupling</th>
</tr>
</thead>
<tbody>
<tr>
<td>Content coupling</td>
<td>No coupling</td>
<td>Modification coupling</td>
</tr>
<tr>
<td>External coupling</td>
<td>Specified coupling</td>
<td>Refinement coupling</td>
</tr>
<tr>
<td>(read only variables)</td>
<td>Scattered coupling</td>
<td>Extension coupling</td>
</tr>
<tr>
<td>Control coupling</td>
<td>Hidden coupling</td>
<td>No Inheritance</td>
</tr>
<tr>
<td>Stamp coupling</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(use all elements)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Data coupling</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5. Check for unacceptable levels of cohesion (i.e. weak cohesion). Refer to the following table:
This Design Artifact Review Guide must be read in conjunction with the Catalogue of Design Errors, where more information about each error can be found.

REQUIREMENTS ERRORS IN DESIGN ARTIFACTS

8. Check the completeness of the requirements in the design artifact.
9. Check the consistency of the requirements in the design artifact.
10. Check the clarity of the requirements in the design artifact.
11. Check the correctness of the requirements in the design artifact.

Other Possible Requirements Errors

12. Check the relevance of the stated requirements in the design artifact.
13. Check the conciseness of the stated requirements in the design artifact.

1. Check namespaces.

Examine the nomenclature used to name various programming elements, such as classes, instance variables, methods, labels, and packages. Identify and examine programming elements in the same or in different classes that share the same name.

2. Check that every class establishes its own invariants.

Identify and examine all available classes and check if invariants are specified, coded and enforced during the lifetime of instances of the classes.

3. Check that increment and decrement operators have been properly used.

Identify and examine statements of code where increment and decrement operators have been used.

4. Check if superclass constructors invoke overridden methods.

Identify and examine inheritance hierarchies where class constructors invoke methods that have been overridden in the subclasses.

5. Check classes that implement more than one interface.

Identify and examine classes that implement more than one interface, when the interfaces belong to disjunct libraries.

6. Check that all instance variables of objects are properly initialised.

Identify and examine code where objects are constructed. Ensure that the constructors that are used initialise all instance variables of the object that they construct to known values.

7. Check instance variables that are accessed directly using object references.

Identify and examine code where instance variables are accessed or modified directly using object references.

8. Check for cast-down-inheritance hierarchy statements.

Identify and examine statements where casting occurs. Ensure that subclass references do not refer/point to superclass objects.

9. Check uses of == and equals().

Identify and examine where operator == and method equals() are used. Ensure they have been used properly.
10. Check uses of inner classes.

Identify and examine inner class structures that attempt to access and modify instance variables of their nesting/outer/enclosing classes.

11. Check for code that has the potential to cause memory leaks.

Identify and examine sections of code as follows:

i) Large objects are used (e.g. large arrays, hashtables, vectors, etc.).
ii) Abstract Windowing Toolkit objects are used.
iii) Third party libraries are used.
iv) Objects that are accessed from static structures are used.

12. Check for code that has the potential to cause thread deadlocks.

Identify and examine code where two or more threads use or attempt to use two or more lock objects.

This Code Artifact Review Guide must be read in conjunction with the Catalogue of Code Errors, where more information about each error can be found.

REQUIREMENTS ERRORS IN CODE ARTIFACTS

14. Check the completeness of the requirements in the design artifact.

15. Check the consistency of the requirements in the design artifact.

16. Check the clarity of the requirements in the design artifact.

17. Check the correctness of the requirements in the design artifact.

Other Possible Requirements Errors

18. Check the relevance of the stated requirements in the design artifact.

19. Check the conciseness of the stated requirements in the design artifact.

DESIGN ERRORS IN CODE ARTIFACTS

20. Check for flaws in the correctness of algorithms.

21. Check the interfaces of the interacting objects.

22. Check the correctness of the inheritance hierarchies constructed by the developer for reuse errors.

23. Check for unacceptable levels of coupling (i.e. strong coupling).

24. Check for unacceptable levels of cohesion (i.e. weak cohesion).
2.3 Training Instruments

2.3.1 Error Framework and Catalogue of Errors Training Instruments

Week 1 Lecture Slides ......................................................... 502
Week 2 Lecture Slides ......................................................... 508
Week 4 Lecture Slides ......................................................... 513
Week 8 Lecture Slides ......................................................... 518
Week 9 Lecture Slides ......................................................... 524

Week 1 Lecture Slides

The Catalogue of Errors Project

- Objectives
- To understand goals of the Catalogue of Errors Projects
- To understand the concept of the Error Framework as the basis for the development of the Catalogue of Errors
- To understand process of the Catalogue of Errors research exercise

The Catalogue of Errors Project: An Introduction

- Targeting errors in software development
  - Preventive approach
    - Preventing errors from being introduced into software
    - Discovering errors early in the lifecycle
  - Detective approach
    - Discovering errors after software is implemented
The Catalogue of Errors Project: The Goals

Project Based on hypothesis that error awareness may help increase Developer proficiency by:

- Minimising error introduction into software
- Improve the ability of developers to correct errors early in software development
- Increase programmer productivity

The Catalogue of Errors Project: The Error Framework

- Error Framework
  - Identity Perspectives
    - Error Name
    - Error Severity
  - Causal Perspectives
    - Error Origin
    - Error Cause
    - Error Prevention
  - Diagnostic Perspectives
    - Error Symptom
    - Error Trigger
The Catalogue of Errors: The Research Exercise

- Two groups of students
  - Mt Lawley Group
    - Receive periodic Catalogue of Errors Training until week 12
  - Joondalup Group
    - Receive extensive Catalogue of Errors Training on week 13
The Catalogue of Errors Project: Involvement Benefits

- Theoretical and practical aspects of how to detect and prevent errors during software development
- Experience the steps to be undertaken to develop a testing and error prevention methodology
- Improve their software development proficiency, including, error prevention, detection, and correction ability and productivity.
- Skills on developing error-free software
- Findings reported at a school level seminar

The Catalogue of Errors Project: Involvement Risks

- None identified
- Open to suggestions
The Catalogue of Errors Project: The Results

- Research results vs Individual Results
  - Research results are based on individual results
  - Individual authorisation/consent
  - Withdrawal of consent
  - Consent Forms
  - De-identification of individual results
  - Findings will be reported at a school level seminar

The Consent Form

**CONSENT FORM**

Project Title: Catalogue for Testing (CATTest) and evaluating its impact in the software development lifecycle

I have read the information above (or, have been informed about all aspects of the above research project) and any questions I have asked have been answered to my satisfaction.

I agree to consent the use of the results of my project for this research activity, knowing I may withdraw such consent at any time.

I agree that the research data gathered for this study may be published provided I am not identifiable. (or, understanding that I may be included)

Patient's name or name of representative ____________________________ Date ____________

Investigator Initials TRUSHAR, ESC/SS ____________________________ Date ____________
The Catalogue of Errors Project: Queries

- Dr Jim MILLAR (Supervisor)
  - Phone: (08) 9370 6547
  - Email: j.millar@cowan.edu.au
- Indrit TROSHANI (Investigator)
  - Phone: (08) 9370 6213
  - Room 13.126
  - Email: i.troshani@ecu.edu.au
Week 2 Lecture Slides

Error Framework & Catalogue of Requirements Errors

- Objectives
  - To understand the perspectives of the Error Framework
  - To understand the categories of the Catalogue of Errors
  - To understand The Catalogue of Requirements Errors

Error Framework

- Identity Perspectives
- Context Perspectives
- Dynamic Perspectives
  - Host
  - Sequence
  - Origin
  - Cause
  - Prevention
  - Symptom
  - Trigger
Identity Perspectives

- **Name**
  - Uniquely identifies the error.
  - Describes the underlying error and provides illustrating example.

- **Severity**
  - Seriousness of errors from the user's point of view and developer's point of view.
  - Range: Critical/Serious/Medium
  - Arbitrary assignment of severity to errors in catalogue

Causal Perspectives

- **Origin**
  - Phase of origin
  - Activity/Task
Causal Perspectives (cont...)

- **Cause**
  - Describes the underlying cause of the error.
  - Examples
    - incomplete or erroneous specification
    - inconsistent method interface
    - ambiguous or inconsistent HCI, etc.

- **Prevention Guidelines**
  - Do not allow the error to be injected in software
  - Ensure the error does not outlive its introduction phase

<table>
<thead>
<tr>
<th>Requirements Phase</th>
<th>Design Phase</th>
<th>Coding Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Definition of goals of software</td>
<td>Identification of classes</td>
<td>Definition of data types</td>
</tr>
<tr>
<td>Definition of constraints</td>
<td>Identification of attributes</td>
<td>Other initialization</td>
</tr>
<tr>
<td>Definition of outcomes</td>
<td>Identification of methods</td>
<td>Software testing</td>
</tr>
<tr>
<td>Definition of viewpoints</td>
<td>Inheritance, aggregation and association definition</td>
<td>Integration</td>
</tr>
<tr>
<td>Importance of goals into requirements</td>
<td>Identification of instructions</td>
<td>Documentation</td>
</tr>
<tr>
<td>Description of requirements</td>
<td>Definition of algorithms</td>
<td>Inspection and testing</td>
</tr>
<tr>
<td>Definition of requirements</td>
<td>Validation criteria</td>
<td>Inspection</td>
</tr>
<tr>
<td>Validation criteria</td>
<td>Inspection</td>
<td></td>
</tr>
<tr>
<td>Inspector</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

510
Diagnostic Perspectives

- **Symptom**
  - Effects by which an error is detected or located.
- **Trigger**
  - Event(s) that help(s) the symptom to be observed.

Catalogue of Requirements Errors

- **Omission**
  - Necessary information omitted from artifact
- **Inconsistency**
  - Information within one part of the artifact contradicts other information in same artifact
- **Incorrect Fact**
  - Information in artifact conflicts with general domain knowledge
- **Ambiguity**
  - Information in artifact is subject to multiple interpretations
- **Extraneous Information**
  - Information is provided that is not needed or used
- **Others**
Omissions

- Goals
  - Subgoals
  - Goal conflicts
  - Goal trees/matrices
- Operationalisation of goals into requirements
  -- Goal Requirements traceability Matrix
- Example (Refer to Catalogue of Requirements Errors)

Inconsistency/Ambiguity/Incorrect Fact

- Viewpoints
  -- Conflicts
- Scenarios
  -- Viewpoint Scenario Traceability Matrix
- Example: Refer to Catalogue of Requirements Errors
Catalogue of Design Errors

- Objectives
  - Identify and understand the following Design Errors
    - Algorithmic
    - Interface
    - Reuse
    - Strong Coupling
    - Weak Coupling

Algorithmic Error

- Includes problems in the design and logic of an algorithm used to solve the problem at hand.
- To prevent:
  - Check logic
  - Establish algorithm preconditions
    - assertions of what should be true before the algorithm starts
  - Establish algorithm postconditions
    - assertions of what should be true after the algorithm ends
Algorithmic Error

```java
public float computeSalary(int noHrs, float hrRate) {
    return salary;
}
```
Preconditions: noHrs >= 0; hrRate >= 0
Postconditions: salary >= 0;

Interface Error

- Two objects interact by one object (the client) calling operations from the other object (the server)
  - Does the client call the server's operations in the "legal" manner?
  - Does the client always interpret the server's answers properly?
- To prevent, inspect design and ensure that:
  - right arguments (number, order, type) are passed in;
  - right values/references are returned.
Reuse Error

- Typically occurs when
  - a derived class modifies a data member of the base class; or
  - inherited methods are re-used in the subclass's new environment

- To prevent
  - ensure reusable class is domain independent
  - specification can be reused in addition to implementation

Reuse Error (cont...

```java
class BaseClass {
  int inherited(int x)
  int redefined() // returns a an int range [1..10]
}
class DerivedClass extends BaseClass {
  int redefined() //returns a an int range [0..70]
  int inherited(int x) //inherited from the base class
}
if inherited(int x) contains this algorithm:
  if(x<0) {
    x = x/redefined();
    return x;
  }
```
Strong Coupling Error

- Strong coupling occurs when a class is largely dependent on other classes.
- Sometimes difficult to achieve.
- To prevent: Minimise information interchange between classes.
- Example: Class A needs a service from class B. This dependency can be eliminated by duplicating the functionality of B in A.

Weak Cohesion Error

- Weak cohesion classes have methods and attributes which have no meaningful relationship to each other.
- To prevent:
  - ensure that methods that belong to a class either access or modify data belonging to that class.
  - Analyse classes and split them into 2 or more, if their methods are not closely related.
Weak Cohesion Error (cont...)

class Employee {
    String name;
    String address;
    Date birthDate;
    Date hireDate;

    ...
    void computeAge() {...}
    void computeSalary() {...}
    int computeCompanyRevenue(Project p) {...}
}

517
Catalogue of Code Errors (Part I)

- Objectives
  - Identify and understand the following six code errors
    - Abuse of namespaces errors
    - Failure to establish class invariants errors
    - Increment and Decrement Operators errors
    - Constructor invoked overridden method errors
    - Misusing contracts of methods inherited by interfaces errors
    - Object initialisation errors

Abuse of namespaces errors

- Names are used to refer to an entity declared in a Java program.
- Java uses six different name spaces
  - package/class/field/method/local variables/label names
- Java uses the context of the name to determine what name it is (e.g. x.f=3)
- Name spaces are designed to give the programmer flexibility, but may be abused.
Abuse of namespaces errors

class point {
    int x; int y;
    public: point(int x, int y) { x = x; y = y; }
}

OR

package reuse;
class reuse {
    reuse reuse (reuse reuse) {
        reuse
        for (int i) {
            if (reuse.use(reuse) == reuse)
                break reuse;
            return reuse;
        }
    }
}

Failure to establish class invariants errors

- An invariant is a condition imposed on the fields of the class.
- Class invariant must remain true throughout the lifespan of the object.
- Failure to impose class invariant may cause the object to assume incorrect states, resulting in an error.
Failure to establish class invariants errors

class purse {
    private int dollars;
    private int cents;
    public void insert(int dollars, int cents) { // code to insert
    } // code to remove
    public static void main(String[] args) {
        Purse purse = new Purse();
        purse.insert(6, 50);
        purse.remove(1000, 50);
    }
}

Increments & Decrement Operator Error

- These errors result from mainly due to lack of knowledge and understanding.
- E.g. i++ is equivalent to i = i + 1.
- They differ in the fact that
  - in i++, i is evaluated only once;
  - in i=i+1, i is evaluated twice;
Increments & Decrement Operator
Error

class x {
    int i = 0;
    private int where() { return i++1; }
    //public firstMethod() { ... ++arr[where()]; ... }
    //public secondMethod() { arr[where()] = arr[where()] + 1; }
}

Constructor invoked overridden method error

class super {
    Super() { printThree(); }
    void printThree { System.out.println("Three"); }
}
class Test extends super {
    int x = (int) Math.PI; //returns the value 3
    public static void main(string args[]) {
        Test t = new Test();
        t.printThree();
    }
    void printThree() { System.out.println(x); }
}
Honouring contracts of methods inherited by interfaces

interface CardDealer {
    void draw();
    void deal();
    void shuffle();
}

interface GraphicalComponent {
    void draw(); // render on default device
    void retain();
    void fillcolor(Color c);
}

interface GraphicalCardDealer extends CardDealer, GraphicalComponent {

}

Difficult to write an implementation for void draw()
Object initialisation errors

```java
class MyDate { // int month, int day, int year
    public MyDate() {} // new MyDate()
    MyDate t конструкция = new MyDate();
    System.out.println(t конструкция);// will print 0/0/0
    System.out.println(t конструкция()); // will print 0/7/0
```
```
Catalogue of Code Errors (Part 2)

- Objectives
  - Identify and understand the following code errors
    - Accessing methods vs Accessing instance variables
    - Cast down the inheritance hierarchy
    - Confusing operator == with method equals()
    - Issues with inner classes
    - Memory Leaks
    - Thread Deadlocks

Accessing Methods vs Accessing Instance Variables

- If a method is invoked through an object reference, the ACTUAL class of the object governs which implementation is used.
- If an instance variable is accessed, the DECLARED class of the reference is used, not the actual class.
Accessing Methods vs Accessing Instance Variables

class SuperShow {
    public String str = "SuperStr";
    public void show() { System.out.println("Super.show":"str"); }
}

class ExtendShow extends SuperShow {
    public String str = "Extended";
    public void show() { System.out.println("Extend.show":"str");
    public static void main(String args[]) {
        ExtendShow ext = new ExtendShow();
        SuperShow sup = ext;
        sup.show(); ext.show();
        System.out.println("SupStr = " + sup.str);
        System.out.println("ExtStr = " + ext.str);
    }
}

The diagram below illustrates the relationships between the SuperShow and ExtendShow classes:

- The SuperShow class has a string attribute `str` and a `show()` method.
- The ExtendShow class extends SuperShow and adds a different string attribute `str` and a `show()` method.
- When an object of ExtendShow is created, it can access both `show()` methods.
- The `show()` methods print the respective strings.

This diagram helps visualize how accessing methods (show()) can be used to access instance variables (str) in a hierarchical class structure.
Cast Down the Inheritance Hierarchy

- In Java it is always acceptable to cast a subclass reference to a superclass reference:
  - SuperClass s = new Subclass(); // 'casting up'
- Superclass references can also be converted to subclass references provided that the reference is properly cast.
  - Superclass super = new Superclass();
  - Subclass s = (Subclass)super; // 'casting down'
- Errors may occur if subclass reference is still pointing to a superclass object.

```java
class Manual { /* code for Manual class */
    class Whale extends Manual { /* code for Whale class */
        class Test {
            public static void main(String args[]) {
                Manual m = new Manual();
                Whale w;
                w = (Whale)m; // explicit class, compiler accepts it.
            }
        }
    }
}

java.lang.ClassCastException will be thrown at run-time.
```
Confusing operator == with method equals()

- Operator == tests that the entities on both of its sides are the same.
- Method equals() tests that the values of both objects are the same.

Confusing operator == with method equals()

```java
class Test {
    public static void main(String args[]) {
        Integer ia = new Integer(10);
        Integer ib = new Integer(10);
        System.out.println("ia==ib is " + (ia==ib)); //"is false" because ia and ib are two different objects
        //error hazard
        System.out.println("ia.equals(ib) " + (ia.equals(ib))); //"is true"
    }
}
```
Issues with Inner Classes

- Inner classes are part of the nesting/enclosing class's implementation.
- This entitles them with the right to have access to enclosing class's private members.
- An enclosing class may have multiple instance of an inner class.
- The above may lead to error hazards if used inadequately.

```java
public class OutterOne {
    private int x;
    public int getX() {return x;}
    public class InnerOne {
        int y;
        public void innerMethod() { // error hazard
            System.out.println("enclosing x is: " + x);
            System.out.println("y is: " + y);
        }
        public void outerMethod() {System.out.println("x is: "+x);}}
    public void outerMethod() {System.out.println("x is: "+x);}}
    public static void main(String args[]){
        OutterOne.InnerOne innerOne = outorOne.new InnerOne();
        innerOne.innerMethod();
        OutterOne.InnerOne innerTwo = outorOne.new InnerOne();
        innerTwo.innerMethod();
        OutterOne.InnerOne innerThree = outorOne.new InnerOne();
        innerThree.innerMethod();
        System.out.println("x = ' + outorOne.getX());}}
```
Memory Leaks

- A memory leak is a situation where the application has run out of memory.
- The existence of the garbage collector does NOT guarantee that memory leaks will not occur.
- The Java Virtual Machine will decide when to run the garbage collector.
- To avoid ensure that:
  - Objects do NOT outstay the time interval the program needs them. Mark them for garbage collection by setting their reference to null.
- Dangerous when large objects (databases) are used

```java
class Example {
    public static void main(String[] args) {
        int bigArray[] = new int[100000];

        // this queries bigArray and obtains a result
        int result = compute(bigArray);

        // at this point bigArray is no longer needed. However,
        // it will be garbage collected only when the
        // method returns. Because bigArray is a local variable,
        // the method, however, will never return, because it
        // loops infinitely, handling the user's input.
        for (;;) { handleInput(result); }
    }
}
```
Thread Deadlocks

- Thread A obtains lock on object X.
- Thread B obtains lock on object Y.
- Thread A needs object Y before it can release object X.
- Thread B needs object X, before it can release object Y.
- Thread A and B, wait for each other indefinitely.
- Application hangs

Thread Deadlocks: Example

class Friendly {
    private String name;
    public synchronized hi() {
        System.out.println("Hi in " + name);
    }
    public synchronized goodbye() {
        System.out.println("Bye in " + name);
    }
}

public class Main {
    public static void main(String[] args) {
        Friendly f = new Friendly();
        Thread a = new Thread() {
            public void run() {
                f.hi();
            }
        };
        Thread b = new Thread() {
            public void run() {
                f.hi();
            }
        };
        a.start();
        b.start();
    }
}
Thread Deadlock: Example

Thread1 is robert.hug() trying to invoke roberta.hugback()
Thread2 in roberta.hug() trying to invoke robert.hugback()
...
... system hangs...

Thread Deadlocks: Example

class friendly {
    private String name;
    private int order = 0;
    public synchronized hug() {
        while (!first.locked() || !second.locked()) {
            first = partner; second = this;
        }
        try {wait()} catch (InterruptedException e) {
    }
    }
    public synchronized hugback() {
        partner.hugback();
        system.out.println(....);
        notify();
    }
}
Thread Deadlocks: Example

Thread 1 in `robet.hug` trying to invoke `robet.hugback`
Thread 1 in `robet.hugback`
Thread 2 in `robet.hag` trying to invoke `robet.hugback`
Thread 2 in `robet.hugback`
2.4 Ethics Instruments

2.4.1 Statement of Disclosure

Statement of Disclosure

1. Introduction
The development of error-free software is one of the major objectives of the software engineering community and an important software quality assurance exercise. In this context, there exist two different approaches to software testing. Firstly, the detective approach suggests that software is first written (specified, analyzed, designed, and coded) and tested using a testing technique. Secondly, the preventive approach suggests that software is 'tested', statically and/or dynamically, as it is specified, analyzed, designed and coded. An extensive literature review in the area reveals that the preventive approach, although occasionally proclaimed as being better than the detective approach, has not received the attention that it deserves. In addition, the few claims made on behalf of the preventive approach to testing have been supported with speculative and benevolent statements lacking the necessary supportive evidence.

2. Project Description
The objective of this project is to develop a Java Catalogue of Errors and evaluate its impact on the software development life cycle. The whole project is based on the premise that awareness of errors and other useful error-related information may lead to the error being recognized, and therefore, not introduced in the software artifact at hand. In order to confirm this belief, the Error Framework has been developed and it has been used to systematically describe all possible errors in the Catalogue of Errors. To confirm our belief a research exercise will be organized and its objective will be to discover the relationship between two main variables, namely, knowledge of the Catalogue of Errors and ability to write software while minimizing introduction of errors, maximizing early error detection and correction and productivity.

Two groups of students will be involved, as surrogates for programmer practitioners. To ensure internal validity for the experiment, it has been decided that the two groups must be homogeneous in terms of background in software development. Information related to participants' background in software development will be collected via an anonymous questionnaire, which will be distributed prior to the research exercise being conducted. The information obtained in this way will be used to determine whether the groups are homogeneous.

Both groups will develop the same 12 weeks duration software project as part of their Internet and Java Programming Unit (Semester 1 2001). One of the groups (Control Group—Joondalup Group) will go through the development process following an incremental object-oriented development approach. The other group (Experimental Group—Mt Lawley Group), on the other hand, will receive periodic training sessions on the Catalogue of Errors and be provided with the relevant documentation. The Catalogue of Errors and its associated training is considered as a preventive approach to testing. To address the potential problem of knowledge sharing between the two groups, it will be ensured that both groups are selected in two different remote campuses (Mt Lawley and Joondalup).

It will be ensured that no group is unfairly discriminated against by providing both groups with an equal opportunity to acquire knowledge. To meet this objective the following research exercise design will be strictly adhered by:

1. The control group will receive no Catalogue of Errors training and documentation for the duration (12 weeks) of the project.
2. The control group will receive extensive Catalogue of Errors training and provided with its complete documentation for the remaining week (week 13) of the semester.

3. The Experimental group will receive Catalogue of Errors training and provided with its complete documentation for the duration (12 weeks) of the project.

4. Since the results of each participant's project will be used to assess their course work element for the unit (50%), they may be subjected to a moderation exercise, if and only if the differences are due to the testing methodology used.

3. Benefits and Risks
It is believed that this research exercise will benefit the participants in the following areas:
1) Theoretical and practical aspects of how to detect and prevent errors during software development.
2) Experience the in steps to be undertaken to develop a testing and error prevention methodology
3) Improvement in software development proficiency, including error prevention, detection, correction, and productivity.
4) Enhancing skills on developing error-free software
5) Research findings will be reported in a Seminar organised at school level.

No risks have been identified.

4. Research Results versus Individual Project Results
It is emphasized that any information collected on any participant's current performance as measured by the result of the individual project or any information related to a participant's past experience or training on software development, that is useful to this research exercise, will be immediately de-identified. No personal or individual information, nor any information which leads to an individual's identification will be retained in any form of storage. If any such information exists, it will be immediately destroyed.

In addition, every individual has the right to refuse his/her consent to (see Consent Form, below) the use of the individual project results for this research or to withdraw such consent at any time. A participant's decision to withdraw his consent to use the results of his/her project for the intents and purposes of this research will not lead to a participant's current position being prejudiced in any way or form.

5. Inquiries
Any questions concerning the project entitled:

Developing Java Catalogue of Errors and Evaluating its Impact on Software Development

can be directed to: Indrit TROSHANI____________________(Principal Investigator)
of School of Computer and Information Science on (08) 9370 6313 or on
j.troshani@ecu.edu.au

If you have any concerns about the project or would like to talk to an independent person, you may contact

Dr JWL MILLAR ________ on (08) 9370 6547 or on j.millar@ecu.edu.au
2.4.2 Consent Form

Consent Form

CONSENT FORM

Project Title: Developing Java Catalogue for Testing and evaluating its impact in software development.

I have been informed about all aspects of the above research project and any questions I have asked have been answered to my satisfaction.

I agree to consent the use of the results of my project for this research activity, realising I may withdraw such consent at any time.

I agree that the research data gathered for this study may be published provided I am not identifiable.

Participant or authorised representative __________________________ Date ___________

Investigator Indrit TROSHANI (SC15) __________________________ Date ___________
# Appendix C: DATA

## Table of Contents

3.1 SOFTWARE DEVELOPMENT BACKGROUND DATA .............................................................................................................. 538

3.1.1 Software Development Background Questionnaire Normality Test Results ................................................................. 538

3.1.2 Intergroup Comparison Kruskal-Wallis Test Results (ML1 vs JOI vs ML2) .............................................................. 540

3.1.3 ML1 Intergroup Software Development Background Proficiency Indicators ......................................................... 542

3.2 CATALOGUE OF ERRORS EVALUATION QUESTIONNAIRES DATA ................................................................................. 543

3.2.1 Bar Charts of Frequencies of Responses to Agreement Questions in Catalogue of Errors Evaluation Questionnaire ............................................................................................................. 543

3.2.2 Frequencies of Responses to Open-Ended Questions ........................................................................................................... 547

3.3 ERROR AND PRODUCTIVITY DATA ................................................................................................................................ 548

3.3.1 Error Density Data: Normality Test Results ....................................................................................................................... 548

3.3.2 Escape Rates Data: Normality Test Results ...................................................................................................................... 549

3.3.3 Productivity Data: Normality Test Results ....................................................................................................................... 550

3.3.4 Distribution of Design Errors Found in Artifacts of ML1, JOI, and ML2 ........................................................................ 551
### 3.1 Software Development Background Data

#### 3.1.1 Software Development Background Questionnaire Normality Test Results

Table C3.1.1 - Tests of Normality for Software Development Questionnaire Data

<table>
<thead>
<tr>
<th>GROUP</th>
<th>Kolmogorov-Smirnov Statistic</th>
<th>df</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>M1</td>
<td>Procedural Software Development Experience (Excellent/Very Good/Good/Fair/Poor)</td>
<td>.229</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Object-Oriented Software Development Experience (Excellent/Very Good/Fair/Good/Poor)</td>
<td>.235</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Requirements Specification</td>
<td>.237</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Low Level Design</td>
<td>.276</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in High Level Design</td>
<td>.219</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Total Design (High + Low Level Design)</td>
<td>.159</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Coding</td>
<td>.235</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Testing</td>
<td>.251</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Total Development Score (req+test+ids+coding+testing )</td>
<td>.125</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Ada</td>
<td>.466</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in C</td>
<td>.468</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in C++</td>
<td>.479</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Visual Basic</td>
<td>.424</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Oracle</td>
<td>.521</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Pascal</td>
<td>.518</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Cobol</td>
<td>.538</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Other Languages</td>
<td>.521</td>
<td>133</td>
</tr>
<tr>
<td></td>
<td>Total Languages (Ada+C+C+++VB+Oracle+Pascal+Cobol+Other )</td>
<td>.233</td>
<td>133</td>
</tr>
<tr>
<td>JO1</td>
<td>Procedural Software Development Experience (Excellent/Very Good/Good/Fair/Poor)</td>
<td>.332</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Object-Oriented Software Development Experience (Excellent/Very Good/Fair/Good/Poor)</td>
<td>.216</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Requirements Specification</td>
<td>.216</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Low Level Design</td>
<td>.268</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in High Level Design</td>
<td>.250</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Total Design (High + Low Level Design)</td>
<td>.160</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Coding</td>
<td>.189</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Testing</td>
<td>.223</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Total Development Score (req+test+ids+coding+testing )</td>
<td>.139</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Ada</td>
<td>.487</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in C</td>
<td>.450</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in C++</td>
<td>.424</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Visual Basic</td>
<td>.345</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Oracle</td>
<td>.475</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Pascal</td>
<td>.487</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Proficiency in Other Languages</td>
<td>.539</td>
<td>39</td>
</tr>
</tbody>
</table>
The Kolmogorov-Smirnov Test tests the hypothesis that the data are normally distributed. A low significance value (less than 0.05) indicates that the distribution of the data differs significantly from a normal distribution. A high significance value (greater than 0.05) indicates that the distribution of the data does not differ significantly from a normal distribution.

<table>
<thead>
<tr>
<th>Group</th>
<th>Kolmogorov-Smirnov Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statistic</td>
</tr>
<tr>
<td>Total Languages (Ada+C+CPP+VB+Oracle+Pascal+Cobol+Other)</td>
<td>.233</td>
</tr>
<tr>
<td>ML2 Procedural Software Development Experience (Excellent/Very Good/Good/Fair/Poor)</td>
<td>.260</td>
</tr>
<tr>
<td>ML2 Object-Oriented Software Development Experience (Excellent/Very Good/Good/Fair/Poor)</td>
<td>.261</td>
</tr>
<tr>
<td>ML2 Proficiency in Requirements Specification</td>
<td>.230</td>
</tr>
<tr>
<td>ML2 Proficiency in Low Level Design</td>
<td>.258</td>
</tr>
<tr>
<td>ML2 Proficiency in High Level Design</td>
<td>.267</td>
</tr>
<tr>
<td>ML2 Proficiency in Total Design (High + Low Level Design)</td>
<td>.128</td>
</tr>
<tr>
<td>ML2 Proficiency in Coding</td>
<td>.258</td>
</tr>
<tr>
<td>ML2 Proficiency in Testing</td>
<td>.240</td>
</tr>
<tr>
<td>ML2 Total Development Score (reg+ids+test+coding+testing)</td>
<td>.130</td>
</tr>
<tr>
<td>ML2 Proficiency in Ada</td>
<td>.479</td>
</tr>
<tr>
<td>ML2 Proficiency in C</td>
<td>.450</td>
</tr>
<tr>
<td>ML2 Proficiency in C++</td>
<td>.512</td>
</tr>
<tr>
<td>ML2 Proficiency in Visual Basic</td>
<td>.382</td>
</tr>
<tr>
<td>ML2 Proficiency in Oracle</td>
<td>.493</td>
</tr>
<tr>
<td>ML2 Proficiency in Pascal</td>
<td>.479</td>
</tr>
<tr>
<td>ML2 Proficiency in Cobol</td>
<td>.536</td>
</tr>
<tr>
<td>ML2 Proficiency in Other Languages</td>
<td>.578</td>
</tr>
<tr>
<td>Total Languages (Ada+C+CPP+VB+Oracle+Pascal+Cobol+Other)</td>
<td>.219</td>
</tr>
</tbody>
</table>
### Table C3.1.2 – Intragroup Comparison Kruskal-Wallis Test Results (ML1 vs JO1 vs ML2)

<table>
<thead>
<tr>
<th>GROUP</th>
<th>N</th>
<th>Mean Rank</th>
<th>Chi-Square</th>
<th>df</th>
<th>Asymp.Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Procedural Software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development Experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excellent/Very Good/Good/Fair/Poor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>121.36</td>
<td>0.289</td>
<td>2</td>
<td>0.666</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>121.42</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>116.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object-Oriented Software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Development Experience</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Excellent/Very Good/Good/Fair/Poor</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>122.60</td>
<td>0.544</td>
<td>2</td>
<td>0.762</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>118.92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>115.47</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proficiency in Requirements Specification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>123</td>
<td>119.28</td>
<td>0.283</td>
<td>2</td>
<td>0.655</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>121.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>118.59</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proficiency in High Level Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>124.32</td>
<td>2.006</td>
<td>2</td>
<td>0.367</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>107.63</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>119.82</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proficiency in Low Level Design</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>121.24</td>
<td>1.925</td>
<td>2</td>
<td>0.382</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>129.44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>112.04</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proficiency in Total Design (High + Low Level Design)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>122.83</td>
<td>0.586</td>
<td>2</td>
<td>0.746</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>118.58</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>115.22</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proficiency in Coding</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>119.37</td>
<td>0.654</td>
<td>2</td>
<td>0.721</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>125.32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>116.99</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proficiency in Testing</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>117.76</td>
<td>0.418</td>
<td>2</td>
<td>0.812</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>120.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>123.96</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total Development Score (req+ides+coding+testing)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>120.43</td>
<td>0.218</td>
<td>2</td>
<td>0.897</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>123.44</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>117.15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>GROUP</td>
<td>N</td>
<td>Mean Rank</td>
<td>Chi-Sq</td>
<td>df</td>
<td>Asymp. Sig</td>
</tr>
<tr>
<td>-------------------</td>
<td>-----</td>
<td>-----------</td>
<td>----------</td>
<td>------</td>
<td>------------</td>
</tr>
<tr>
<td><strong>Proficiency in Ada</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>121.54</td>
<td>0.362</td>
<td>2</td>
<td>0.835</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>119.49</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>117.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proficiency in C</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>118.15</td>
<td>0.369</td>
<td>2</td>
<td>0.832</td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>122.21</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>122.39</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proficiency in C++</strong></td>
<td></td>
<td></td>
<td>4.845</td>
<td>2</td>
<td>0.089</td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>120.45</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>133.33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>111.34</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proficiency in Visual Basic</strong></td>
<td></td>
<td></td>
<td>1.330</td>
<td>2</td>
<td>0.514</td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>116.80</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>118.78</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>127.06</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proficiency in Oracle</strong></td>
<td></td>
<td></td>
<td>2.977</td>
<td>2</td>
<td>0.226</td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>115.77</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>128.08</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>123.69</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proficiency in Pascal</strong></td>
<td></td>
<td></td>
<td>2.816</td>
<td>2</td>
<td>0.245</td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>115.67</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>124.01</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>126.25</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proficiency in Cobol</strong></td>
<td></td>
<td></td>
<td>5.326</td>
<td>2</td>
<td>0.070</td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>118.70</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>116.00</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>124.92</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Proficiency in Other Languages</strong></td>
<td></td>
<td></td>
<td>1.822</td>
<td>2</td>
<td>0.400</td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>122.27</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>113.13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>119.49</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total Languages</strong></td>
<td>(Ada+C+++VB+Oracle+Pascal+Cobol+Other)</td>
<td></td>
<td>1.622</td>
<td>2</td>
<td>0.445</td>
</tr>
<tr>
<td>ML1</td>
<td>133</td>
<td>115.16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JO1</td>
<td>39</td>
<td>127.36</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ML2</td>
<td>67</td>
<td>125.33</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>239</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Kruskal-Wallis test is a non-parametric test. Significance levels below .05 indicate that the group locations differ, whereas significance levels above .05 indicate that the group locations are not significantly different.
### 3.1.3 ML1 Intragroup Software Development Background Proficiency Indicators

Table C3.1.3 – Descriptive Statistics of ML1 Software Development Proficiency Indicators

<table>
<thead>
<tr>
<th>Requirement Proficiency</th>
<th>N</th>
<th>Valid</th>
<th>Missing</th>
<th>133</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>133</td>
</tr>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td>2.6992</td>
</tr>
<tr>
<td>Median</td>
<td></td>
<td></td>
<td></td>
<td>3.0000</td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
<td>4.00</td>
</tr>
<tr>
<td>Minimum</td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td></td>
<td></td>
<td></td>
<td>5.00</td>
</tr>
<tr>
<td>Percentiles</td>
<td>25</td>
<td></td>
<td></td>
<td>2.0000</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td>3.0000</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td></td>
<td></td>
<td>3.0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Average Design Proficiency (High+Low)/2</th>
<th>N</th>
<th>Valid</th>
<th>Missing</th>
<th>133</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td>2.5526</td>
</tr>
<tr>
<td>Median</td>
<td></td>
<td></td>
<td></td>
<td>2.5000</td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
<td>3.50</td>
</tr>
<tr>
<td>Minimum</td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td></td>
<td></td>
<td></td>
<td>4.50</td>
</tr>
<tr>
<td>Percentiles</td>
<td>25</td>
<td></td>
<td></td>
<td>2.0000</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td>2.5000</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td></td>
<td></td>
<td>3.0000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Code Proficiency</th>
<th>N</th>
<th>Valid</th>
<th>Missing</th>
<th>133</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean</td>
<td></td>
<td></td>
<td></td>
<td>2.7068</td>
</tr>
<tr>
<td>Median</td>
<td></td>
<td></td>
<td></td>
<td>3.0000</td>
</tr>
<tr>
<td>Range</td>
<td></td>
<td></td>
<td></td>
<td>4.00</td>
</tr>
<tr>
<td>Minimum</td>
<td></td>
<td></td>
<td></td>
<td>1.00</td>
</tr>
<tr>
<td>Maximum</td>
<td></td>
<td></td>
<td></td>
<td>5.00</td>
</tr>
<tr>
<td>Percentiles</td>
<td>25</td>
<td></td>
<td></td>
<td>2.0000</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td></td>
<td></td>
<td>3.0000</td>
</tr>
<tr>
<td></td>
<td>75</td>
<td></td>
<td></td>
<td>3.0000</td>
</tr>
</tbody>
</table>
3.2 Catalogue of Errors Evaluation Questionnaires Data

3.2.1 Bar Charts of the Frequencies of Agreement Questions in Catalogue of Errors Evaluation Questionnaire

Figure C3.2.1 (A (i, ii, iii, iv, v)) – Bar Charts of Agreement Questions Responses for the Catalogue of Requirements Errors
Figure C3.2.1 (B (i, ii, iii, iv, v)) – Bar Charts of Agreement Questions Responses for the Catalogue of Design Errors
Figure C3.2.1 (B (i,ii,iii,iv,v)) – Bar Charts of Agreement Questions Responses for the Catalogue of Code Errors
<table>
<thead>
<tr>
<th>R1</th>
<th>D1</th>
<th>C1</th>
<th>The Catalogue of Requirements/Design/Code Errors is easy to use.</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>D2</td>
<td>C2</td>
<td>The Catalogue of Requirements/Design/Code Errors is easy to follow.</td>
</tr>
<tr>
<td>R3</td>
<td>D3</td>
<td>C3</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand requirements/design/code errors.</td>
</tr>
<tr>
<td>R4</td>
<td>D4</td>
<td>C4</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand the origin of some requirements/design/code errors.</td>
</tr>
<tr>
<td>R5</td>
<td>D5</td>
<td>C5</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped appreciate the severity of some requirements/design/code errors.</td>
</tr>
<tr>
<td>R6</td>
<td>D6</td>
<td>C6</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand the causes of some requirements/design/code errors.</td>
</tr>
<tr>
<td>R7</td>
<td>D7</td>
<td>C7</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand how to prevent some requirements/design/code errors.</td>
</tr>
<tr>
<td>R8</td>
<td>D8</td>
<td>C8</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me understand the symptoms of some requirements/design/code errors.</td>
</tr>
<tr>
<td>R9</td>
<td>D8</td>
<td>C9</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me identify triggers to expose some requirements/design/code errors.</td>
</tr>
<tr>
<td>R10</td>
<td>D10</td>
<td>C10</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me identify some requirements/design/code errors.</td>
</tr>
<tr>
<td>R11</td>
<td>D11</td>
<td>C11</td>
<td>The Catalogue of Requirements/Design/Code Errors has helped me correct possible requirements/design/code errors.</td>
</tr>
<tr>
<td>R12</td>
<td>D12</td>
<td>C12</td>
<td>The Catalogue of Requirements/Design/Code Errors has improved my skills to specify higher quality requirements/design/code.</td>
</tr>
</tbody>
</table>
### 3.2.2 Frequencies of Responses to Open-Ended Questions

#### Table C3.2.3 – Frequencies of Responses to Open-Ended Questions

<table>
<thead>
<tr>
<th>No</th>
<th>Recommendations</th>
<th>Frequency</th>
<th>ML1</th>
<th>ML2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>JCATT Req.</td>
<td>JCATT Design</td>
<td>JCATT Code</td>
</tr>
<tr>
<td></td>
<td></td>
<td>n=133</td>
<td>n=133</td>
<td>n=26</td>
</tr>
<tr>
<td>1</td>
<td>Clarification of terms</td>
<td>25</td>
<td>13</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td></td>
<td>18.80%</td>
<td>9.77%</td>
<td>4.76%</td>
</tr>
<tr>
<td>2</td>
<td>More examples</td>
<td>30</td>
<td>26</td>
<td>12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>22.56%</td>
<td>9.52%</td>
<td>9.95%</td>
</tr>
<tr>
<td>3</td>
<td>More time to be spent in class</td>
<td>16</td>
<td>13</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12.03%</td>
<td>9.77%</td>
<td>7.94%</td>
</tr>
<tr>
<td>4</td>
<td>Improve Layout/Font</td>
<td>4</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3.01%</td>
<td>0.79%</td>
<td>1.49%</td>
</tr>
<tr>
<td>5</td>
<td>More Detail</td>
<td>2</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1.59%</td>
<td>0.0%</td>
<td>2.99%</td>
</tr>
<tr>
<td>6</td>
<td>Less Detail</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.75%</td>
<td>0.79%</td>
<td>2.99%</td>
</tr>
<tr>
<td>7</td>
<td>Include additional errors</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
</tr>
<tr>
<td>8</td>
<td>Include syntax errors</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0%</td>
<td>1.59%</td>
<td>0.0%</td>
</tr>
<tr>
<td>9</td>
<td>Simplify Error Description</td>
<td>0</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.0%</td>
<td>1.59%</td>
<td>1.49%</td>
</tr>
<tr>
<td>10</td>
<td>Internet Accessibility</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.75%</td>
<td>0.79%</td>
<td>0.0%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0.75%</td>
<td>0.79%</td>
<td>0.0%</td>
</tr>
</tbody>
</table>

#### Commendations

| No | JCATT is sufficiently detailed                      | 12        | 2              | 0              |
|    |                                                      | 9.02%     | 1.50%          | 0.0%          |
|    |                                                      | 4         | 0              | 2              |
|    |                                                      | 3.01%     | 3.01%          | 0.0%          |
|    |                                                      | 10        | 11             | 9              |
|    |                                                      | 7.52%     | 8.27%          | 6.87%         |
|    |                                                      | 0         | 0              | 0              |
|    |                                                      | 0.0%      | 0.0%           | 2.38%         |

| No | No further improvement needed                        | 4         | 4              | 0              |
|    |                                                      | 3.01%     | 3.01%          | 0.0%          |
|    |                                                      | 9         | 3              | 0              |
|    |                                                      | 7.52%     | 8.27%          | 6.87%         |
|    |                                                      | 0         | 0              | 0              |
|    |                                                      | 0.0%      | 0.0%           | 2.38%         |

| No | JCATT is useful to prevent errors                    | 10        | 11             | 9              |
|    |                                                      | 7.52%     | 8.27%          | 6.87%         |
|    |                                                      | 0         | 0              | 0              |
|    |                                                      | 0.0%      | 0.0%           | 2.38%         |

| No | JCATT has sufficient examples                        | 0         | 0              | 3              |
|    |                                                      | 0.0%      | 0.0%           | 2.38%         |
|    |                                                      | 0         | 0              | 0              |
|    |                                                      | 0.0%      | 0.0%           | 2.38%         |
3.3 Error and Productivity Data

3.3.1 Error Density Data: Normality Test Results

Table C3.3.1 - Tests of Normality for Density of Errors Data

<table>
<thead>
<tr>
<th>GROUP</th>
<th>Kolmogorov-Smirnov</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Statistic</td>
</tr>
<tr>
<td>ML1</td>
<td>Density of Requirements Errors</td>
</tr>
<tr>
<td></td>
<td>Density of Other Requirements Errors (Other Req Errors/Number of Pages of Requirements Artifact)</td>
</tr>
<tr>
<td></td>
<td>Density of Design Errors</td>
</tr>
<tr>
<td></td>
<td>Density of Other Design Errors (Other Design Errors/Number of Pages in Design Artifact)</td>
</tr>
<tr>
<td></td>
<td>Density of Coding Errors (Total Code Errors/KLOC)</td>
</tr>
<tr>
<td></td>
<td>Density of Other Code Errors (Other Code Errors/KLOC)</td>
</tr>
<tr>
<td>JO1</td>
<td>Density of Requirements Errors</td>
</tr>
<tr>
<td></td>
<td>Density of Other Requirements Errors (Other Req Errors/Number of Pages of Requirements Artifact)</td>
</tr>
<tr>
<td></td>
<td>Density of Design Errors</td>
</tr>
<tr>
<td></td>
<td>Density of Other Design Errors (Other Design Errors/Number of Pages in Design Artifact)</td>
</tr>
<tr>
<td></td>
<td>Density of Coding Errors (Total Code Errors/KLOC)</td>
</tr>
<tr>
<td></td>
<td>Density of Other Code Errors (Other Code Errors/KLOC)</td>
</tr>
<tr>
<td>ML2</td>
<td>Density of Requirements Errors</td>
</tr>
<tr>
<td></td>
<td>Density of Other Requirements Errors (Other Req Errors/Number of Pages of Requirements Artifact)</td>
</tr>
<tr>
<td></td>
<td>Density of Design Errors</td>
</tr>
<tr>
<td></td>
<td>Density of Other Design Errors (Other Design Errors/Number of Pages in Design Artifact)</td>
</tr>
<tr>
<td></td>
<td>Density of Coding Errors (Total Code Errors/KLOC)</td>
</tr>
<tr>
<td></td>
<td>Density of Other Code Errors (Other Code Errors/KLOC)</td>
</tr>
</tbody>
</table>

The Kolmogorov-Smirnov Test tests the hypothesis that the data are normally distributed. A low significance value (less than 0.05) indicates that the distribution of the data differs significantly from a normal distribution. A high significance value (greater than 0.05) indicates that the distribution of the data does not differ significantly from a normal distribution.
3.3.2 Escape Ratios Data: Normality Test Results

The Kolmogorov-Smirnov Test tests the hypothesis that the data are normally distributed. A low significance value (less than 0.05) indicates that the distribution of the data differs significantly from a normal distribution. A high significance value (greater than 0.05) indicates that the distribution of the data does not differ significantly from a normal distribution.

<table>
<thead>
<tr>
<th>GROUP</th>
<th>% of Requirements Errors Found at Design Artifact</th>
<th>% of Requirements Errors Found at Code Artifact</th>
<th>% of Design Errors Found at Code Artifact</th>
<th>Kolmogorov-Smirnov Test Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML1</td>
<td>.197</td>
<td>.394</td>
<td>.196</td>
<td>Statistic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.000</td>
</tr>
<tr>
<td>JO1</td>
<td>.099</td>
<td>.191</td>
<td>.129</td>
<td>Statistic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.200</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.002</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.136</td>
</tr>
<tr>
<td>ML2</td>
<td>.179</td>
<td>.397</td>
<td>.167</td>
<td>Statistic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.001</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.000</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>.002</td>
</tr>
</tbody>
</table>
3.3.3 Productivity Data: Normality Test Results

Table C3.3.3 Tests of Normality for Productivity Data

<table>
<thead>
<tr>
<th>GROUP</th>
<th>Kolmogorov-Smirnov Statistic</th>
<th>df</th>
<th>Sig.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ML1</td>
<td>Productivity to Develop Requirement's Artifact</td>
<td>.154</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>Productivity to Develop Design Artifact</td>
<td>.101</td>
<td>126</td>
</tr>
<tr>
<td></td>
<td>Productivity to Develop the Code Artifact (LOC/Hours spent on Code Artifact)</td>
<td>.081</td>
<td>126</td>
</tr>
<tr>
<td>JO1</td>
<td>Productivity to Develop Requirement's Artifact</td>
<td>.163</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Productivity to Develop Design Artifact</td>
<td>.113</td>
<td>39</td>
</tr>
<tr>
<td></td>
<td>Productivity to Develop the Code Artifact (LOC/Hours spent on Code Artifact)</td>
<td>.105</td>
<td>39</td>
</tr>
<tr>
<td>ML2</td>
<td>Productivity to Develop Requirement's Artifact</td>
<td>.098</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Productivity to Develop Design Artifact</td>
<td>.161</td>
<td>65</td>
</tr>
<tr>
<td></td>
<td>Productivity to Develop the Code Artifact (LOC/Hours spent on Code Artifact)</td>
<td>.107</td>
<td>65</td>
</tr>
</tbody>
</table>

The Kolmogorov-Smirnov Test tests the hypothesis that the data are normally distributed. A low significance value (less than 0.05) indicates that the distribution of the data differs significantly from a normal distribution. A high significance value (greater than 0.05) indicates that the distribution of the data does not differ significantly from a normal distribution.

550
3.3.4 Distribution of Design Errors Found in Artifacts of ML1, JO1 and ML2

GROUP: 1 ML1

- Other Design Errors: 6.7%
- Weak Cohesion Error: 25.4%
- Algorithmic Error: 8.7%
- Interface Error: 14.7%
- Reuse Error: 4.2%
- Strong Coupling Error: 40.3%

Figure C3.3.4.1 – Distribution of Design Errors Found in ML1 Participants’ Design Artifacts

GROUP: 2 JO1

- Other Design Errors: 4.4%
- Weak Cohesion Error: 30.1%
- Algorithmic Error: 9.8%
- Interface Error: 15.3%
- Reuse Error: 6.6%
- Strong Coupling Error: 33.9%

Figure C3.3.4.2 – Distribution of Design Errors Found in JO1 Participants’ Design Artifacts
Figure C3.3.4.3 – Distribution of Design Errors Found in ML2 Participants’ Design Artifacts

Figure C3.3.4.4 – Distribution of Design Errors Found in ML1 Participants’ Code Artifacts
GROUP: 2 JO1

Weak Cohesion Errs
49.1%

Algorithmic Errors
4.5%

Reuse Errors
2.7%

Strong Coupling Errs
43.6%

Figure C3.3.4.5 – Distribution of Design Errors Found in JO1 Participants’ Code Artifacts

GROUP: 3 ML2

Weak Cohesion Errs
31.1%

Algorithmic Errors
10.4%

Reuse Errors
2.8%

Strong Coupling Errs
55.7%

Figure C3.3.4.6 – Distribution of Design Errors Found in ML2 Participants’ Code Artifacts
Appendix D: Samples of Errors Found in Development Artifacts

1. List of Requirements

1.1. List of Functions

1. Display main screen
2. Lobby design: front desk, telephone, Safe Deposit Room, ledge and wall
3. Create Staff
4. Play background music

Figure D4.1 – Requirement Error (Ambiguity) found in Requirements Artifact

3. User selects “Start” button to start the program.
4. Simulator creates Front Desk Guest.
5. Front Desk Guest walks to the front desk; staff will attend to the Guest.

Figure D4.2 – Requirement Error (Omission) found in Requirements Artifact

Class: Car

Attributes:
- carMoving = Boolean
- carDisplayed = Boolean
- carPosX = The X position of the car on the UI
- carPosY = The Y position of the car on the UI

Behaviours:
- pointComponent: draws the image of a car at a location (coordinates) on the intersection
- moveCar: sets carMoving to true
- stopCar: sets carMoving to false
- removeCar: removes a car - sets carDisplayed to false

Figure D4.3 – Design Error (Weak Cohesion) found in Design Artifact
import java.net.*;
import java.io.*;

public class Customer extends Thread {
  private Lobby theLobby;
  private boolean active = false;
  private int Xoordinate, Ycoordinate, count, random;

  public Customer(Lobby l) {
    this.theLobby = l;
  }

  public void run() {
    // None of these instance variables is initialized.
  }
}

Figure D4.4 - Code Error (Object Initialisation Error) found in Code Artifact

Low Level Design

Method changeTrafficlight in class Synchronizer:

No Parameters in method call.

Begin changeTrafficlight.

If willStatus is false and hasStatus is false /flights in pedestrian cycle
Then
  willStatus = true //change to William St. cycle
  hasStatus = false

else if willStatus is true
Then
  willStatus = false //change to Hay St. cycle
  hasStatus = true

else
  willStatus = false //change to pedestrian cycle.
  hasStatus = false

Call setTrafficphase in class Trafficlight
End changeTrafficlight.

Figure D4.5 - Requirements Error (Omission) found in Design Artifact
Figure D4.6 – Requirement Error (Omission) found in Code Artifact

```java
// when we add an air guest, what to do
add_new_guest();
add_new_guest_button.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        myGuestQueue.add_new_guest(95.5);
        myCabin.set_number_at_95(myGuestQueue.return_queue_size());
    }
});

// when the phone rings

Figure D4.7 – Requirement Error (Omission) found in Code Artifact

```
private Container GUI;

private AudioClip singing, lobbyMusic;

public void init()
|  | setSize( 650, 610);
|  | getContentPane().setLayout( new FlowLayout( ) );

theLobbyPanel = new JPanel();
theStatusPanel = new JPanel();
singing = getAudioClip(getDocumentBase(), "sounds//phone.wav");
lobbyMusic = getAudioClip(getDocumentBase(), "sounds//lobby music.wav");

Figure D4.7 – Design Error (Strong Coupling) found in Code Artifact