1-1-1994

Software metrics for monitoring software engineering projects

Edwin C. Lim

Edith Cowan University

USE OF THESIS

SOFTWARE METRICS FOR MONITORING SOFTWARE ENGINEERING PROJECTS

BY

Edwin LIM Charng Yih

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

Master of Applied Science (Computer Studies)

at the School of Mathematics, Information Technology, and Engineering,
Edith Cowan University

Date of Submission : 25 August 1994

TABLE OF CONTENTS

Chapter 6 : Measuring Software Size

Chapter 7 : Measuring Productivity

Chapter 8 : Measuring Software Quality

Chapter 9 : Measuring Software Usability

Chapter 10 : Final Analysis And Conclusion

ABSTRACT

As part of the undergraduate course offered by Edith Cowan University, the Department of Computer Science has (as part of a year's study) a software engineering group project. The structure of this project was divided into two units, Software Engineering 1 and Software Engineering 2. In Software Engineering 1, students were given the group project where they had to complete and submit the Functional Requirement and Detail System Design documentation. In Software Engineering 2, students commenced with the implementation of the software, testing and documentation. The software was then submitted for assessment and presented to the client.

To aid the students with the development of the software, the department had adopted EXECOM's APT methodology as its standard guideline. Furthermore, the students were divided into groups of 4 to 5, each group working on the same problem. A staff adviser was assigned to each project group.

The purpose of this research exercise was to fulfil two objectives. The first objective was to ascertain whether there is a need to improve the final year software engineering project for future students by enhancing any aspect that may be regarded as deficient. The second objective was to ascertain the factors that have the most impact on the quality of the delivered software.

The quality of the delivered software was measured using a variety of software metrics. Measurement of software has mostly been ignored until recently or used without true understanding of its purpose. A subsidiary objective was to gain an understanding of the worth of software measurement in the student environment.

One of the conclusions derived from the study suggests that teams who spent more time on software design and testing, tended to produce better quality software with less defects. The study also showed that adherence to the APT methodology led to the project being on schedule and general team satisfaction with the project management. One of the recommendations made to the project co-ordinator was that staff advisers should have sufficient knowledge of the software engineering process.

DECLARATION

I certify that this thesis does not incorporate without acknowledgment any material previously submitted for a degree or diploma in any institution of higher education; and that to the best of my knowledge and belief it does not contain any material previously published or written by another person except where due reference is made in the text.

Signature

Date 25/8/94

ACKNOWLEDGEMENTS

Special thanks to my family for their support, patience and understanding - without them I would not be here today. Special thanks also to Foong Choo for her constant encouragement and support.

I would like to take this opportunity to thank my supervisors Dr Jim Millar for providing me with this research project, Dr Ken Mullin and Mr Stuart Hope for their supervision and guidance. Appreciation is due to the 1993 software engineering students for providing me with the necessary data for my research. I would also like to thank all the staff members from the Computer Science Department for their insights - especially Dr Thomas O'Neill. Appreciation is also due to Mr Geoff Barrett for helping me to proof read my work.

LIST OF DIAGRAMS

LIST OF TABLES

LIST OF GRAPHS

CHAPTER 1 : INTRODUCTION

1.1 INTRODUCTION

"Group projects are an important part of many software engineering courses. Factors, such as group dynamics, egoless programming and team organisation, that affect the way programmers work together cannot be taught effectively in a classroom setting" (Calliss et al., 1991, p. 25). To appreciate the dynamics of group behaviour it is essential for students to participate in a group project as this facilitate and enhances their understanding of the solutions to problems experienced in a group project.

As part of the Bachelor of Applied Science (Information Science) course offered by Edith Cowan University, the Department of Computer Science has formulated, in the final year of that course, a software engineering group project. This group project is divided into two units, Software Engineering 1 and Software Engineering 2 and they are offered in semesters one and two respectively. The purpose of the project is to design a piece of software to meet a client's requirements. During semester one, students are required to complete and submit the Functional Requirement and Detail System Design documentation. During semester two, the students undertake the implementation of the software, testing and documentation. The maintenance phase is omitted because it is not feasible within the current course structure.

Each group is required to present its product, whether it is completed or not, before a judging panel that is usually made up of the project co-ordinator, the software engineering unit co-ordinator, the group's staff adviser and the client. Each group is given an hour to present the functionality of their software.

For the past two years, the software engineering project has been a group project. There were 16 groups, and each group consisted of 4 or 5 students. For each group, one student member was appointed project leader and their primary were allocated roles such as programmer, documenter and tester. A staff adviser was also assigned to each project group. The staff adviser was not to have any active role in the project - rather he/she acted as a consultant to the members of each group.

The Computer Science department has adopted EXECOM's APT (1991) methodology as the standard guideline for developing software. Since 1991, students undertaking the software engineering project have applied this methodology. Students had to purchase the licence to use this methodology.

1.1.1 Significance Of The Study

There are two main objectives to this study. The first objective is to ascertain whether there is a need to improve the final year software engineering project for future students by enhancing any aspects that may be regarded as deficient. Some of these aspects are :

- ❑ The software development methodology
- ❑ Arrangements between staff advisers and students
- ❑ Quality of the project
- ❑ Method(s) of conveying user requirements to project groups

The second objective is to ascertain the factors that have the greatest impact on the quality of the delivered software. To achieve this, it is necessary to firstly identify and measure the factors that influence software quality, and secondly measure the software quality itself. Some of the influencing factors are :

- ❑ Quality of project management
    - Project scheduling
    - Risk management
    - Configuration management
- ❑ Availability of hardware, software and meeting rooms
- ❑ Access to client
- ❑ Quality of team work
- ❑ Choice of software

- □ Influence of staff adviser
- □ Usefulness of the APT methodology
- □ Individual attributes
    - Age
    - Gender
    - Experience

The key software quality measures are :

- □ Functionality
- □ Size
- □ Usability
- □ Performance

Having identified and obtained a measure of the influencing factors and software quality, the final step will be to perform a series of statistical analyses to determine which factors have the highest impact on quality and to what degree.

1.1.2 Major Questions To Be Addressed

For the past three years, students undertaking the software engineering project, have been developing software using the students' version of EXECOM's (1991) APT methodology. It contains guidelines on the steps that are required to produce a piece of software. The software that students produced were assessed by the judging panel. Students were then awarded a mark for their effort. The APT methodology is generally accepted by industry in Western Australia but there is not any empirical data as to its usefulness in a university environment. Students were instructed to use this methodology, but were they producing quality software? The questions that will be addressed are:

- □ How useful was the APT methodology, from the students' point of view?
- □ Was it applicable to the type of software and paradigm used by the students?

Each project group was assigned a staff adviser whose role was to act as a consultant to students. In practice, it was not mandatory that students report regularly to their staff adviser. However, the perception was that groups who stayed in close contact with their staff adviser improved their chances of producing better quality software. The questions that will be addressed are :

- □ How did the staff members feel about being assigned to supervise a project group(s)?
- □ Did he/she have sufficient background in the area of software engineering that could be beneficial to the group he/she was supervising?
- □ Was he/she familiar with the software engineering methodology standard adopted?
- □ Did he/she spend sufficient time with the project group to be of any benefit to the students?
- □ Did the staff adviser have a good understanding of what was needed in the proposed system?

Students had two semesters in which to complete the software engineering project. This provided the students with sufficient time to implement the various phases, which included the Functional Requirement, Detail System Design, Coding and Testing. The Maintenance phase was not possible within the current project structure, due to its time constraints, and tnerefore was not expected. The students were required to undertake Project Management tasks such as risk management, configuration management and task scheduling. The project leader within each group was appointed by the members themselves. The questions that will be addressed are :

- □ How much time did a group spend (in total) on the project?
- □ How much time did individual students spend working alone versus working in the group?
- □ How much time was spent on each phase of the software development life-cycle?
- □ Was there peer assessment for each group?
- □ How well was the project managed?
- □ Did every member of the group contribute and, if so, how well was his or her contribution received by the rest of the group?
- □ Were there any internal conflicts among members of a group?

The aim of the project was to provide students with the experience of working in groups and to tackle a problem that was big enough to simulate a "real-world" situation. The major component of the assessment by the judging panel was the software demonstration. The students may be able to deliver working software but there are many other factors involved in regard to the quality of the software. Therefore, the following questions will be addressed :

- □ What was the size of the final product?
- □ How functional was the final product?
- □ How useable was the final product?
- □ How installable was the final product?
- □ What score did the final product get from the judging panel?

1.2 METHOD OF INVESTIGATION

This whole research project, revolves around the software engineering projects. Data will be collected from the students, staff advisers and by evaluating the final product.

1.2.1 Research Methods And Techniques

The first method of gathering data was the use of questionnaires. In total, three questionnaires were prepared. The first was distributed, on a weekly basis, between the period of April 1993 to June 1993. The second questionnaire was distributed, again on a weekly basis, between the period of August 1993 to November 1993. Towards the end of the second semester, a third questionnaire was provided; each student was required to fill in this questionnaire after their group's project demonstration and he/she was asked to supply an estimate of individual effort. Some of the questions asked were similar to those asked in the second set. This is to allow cross-checking of students' responses between the second and third set of questionnaires. The aim of these questionnaires was to gather information on the effort that each student was contributing to the project.

The second method of data gathering was by interviewing the staff advisers. This was to ascertain the relationship between the staff adviser and the students, the adviser's opinion about the whole exercise of supervising a project group, etc. Each interview was structured so that every staff adviser received the same set of questions.

The third method of data gathering was to measure the software metrics of the software produced by each project group. The objective of this exercise was to determine the quality of the delivered software, such as usability, installability, functionality and size of the software.

1.3 ETHICAL ISSUES

Since this research involves individuals, the data gathered will be kept confidential (as required by the **Committee for the Conduct of Ethical Research**). The data gathered will be made known only to the supervisors[1] and the investigator[2]. Students undertaking this research will not be known by name. The only information the investigator has is the student's group number and personal identifier. Information on the staff advisers was restricted to their group allocation number. The data will not be kept after the research is completed. All data recorded in written form will be shredded and those stored on magnetic medium (such as computer floppy diskettes) will be erased.

[1] Dr Ken Mullin, Mr Stuart Hope and Dr Jim Millar of the Department of Computer Science, Mount Lawley Campus.

[2] Edwin LIM Charng Yih (Student Number 0899367)

2.1 SOFTWARE ENGINEERING

Developing a piece of software that satisfies user requirements, on budget and on schedule is every software developers' dream. But in the real world, this is often not the case. Software development projects are often late and exceed their original projected budgets by as much as 100 to 200+%. So, whose fault is this? The fault is usually due to ineffective initial estimates and to the manager's incapacity to accurately monitor the project's progress (Kemerer, 1993, p. 87).

Hence, one major problem that senior computer professionals in charge of project teams face, is to keep effective control on all aspects of the project. The Software Development Life Cycle contains a large software management component covering a range of activities. If these activities are not properly managed, potential errors are bound to occur, resulting in the project exceeding its projected budget and schedule. To manage all aspects of the software development, there must be some form of measuring mechanism. It is common management theory that, "you are not able to manage what you cannot measure" (Grupe et al., 1991, p. 26).

This chapter will focus on the issue of good software engineering practices and specifically on software metrics in project management. To facilitate this, the role of measurement and software metrics will be considered, including their impact on project management. Additionally the various paradigms that are currently available will be discussed. In focusing on good software engineering practices, the role that academic institutions are playing in the area of providing students with theoretical knowledge on not only software engineering but also practical skills in software development, will also be examined.

2.1.1 What is software engineering?

An early definition of software engineering, which is found in the literature, is (Pressman, 1992, p. 23) :

> "The establishment and use of sound engineering principles in order to obtain economically [*sic*] software that is reliable and works efficiently on real machines."

However, developing a piece of software that is "reliable and works efficiently on real machines" is much harder in the real world (Pressman, 1992, p. 23). There are many problems associated with software development. Such problems include late delivery of software, budget over-run, unreliable software, poor maintainability and poor performance (Sommerville, 1989, p. 3). These problems are categorised by many industry observers as a "crisis". Hence the term ***software crisis*** or ***software affliction*** (Pressman, 1992, p. 17), which suggests a set of problems that are encountered in the development of software. These problems are not restricted to software that does not work properly. Rather, the affliction includes problems associated with the development and maintenance of software.

According to Pressman (1992, p. 23), software engineering is an approach to a solution for software affliction that can be achieved by applying specific tasks to "... all phases of software development, using automated tools to aid these tasks, using more powerful building blocks for software implementation, using better techniques for software quality assurance ...", and by enforcing good project coordination, control and management. Software engineering consists of a set of three key elements - methods, tools and procedures. These elements will enable management to "... control the process of software development and provide the practitioner with a foundation for building high-quality software in a productive manner" (Pressman, 1992, p. 24).

The software engineering ***methods*** provide the technical ("*how to's*") steps for building software. The tasks include "project planning and estimation, system and software requirement analysis, design of data structure, program architecture and algorithm procedure, coding, testing and maintenance" (Pressman, 1992, p. 24). It also includes a set of criteria for software quality. The software engineering ***tools*** provide these methods with automated or semi-automated support. Currently, there are tools that will support all the methods mentioned above. All these tools can be integrated so that information created by one tool can be shared among the other tools through a system called CASE (computer-aided software engineering). The software engineering ***procedures*** are what hold the methods and tools together, and "... enable rational and timely development of computer software" (Pressman, 1992, p. 24). These methods, tools and procedures, as a whole, can be viewed as a software development methodology.

A simpler definition provided by the *IEEE Standard Glossary of Software Engineering Terminology* (Vliet, 1993, p. 5) defines software engineering as "the systematic approach to the development, operation, maintenance, and retirement of software".

2.1.2 Software Development Life Cycle

There are currently a number of life-cycle paradigms namely the *classic life cycle* or *waterfall model*, *prototyping*, the *evolutionary model*, the *spiral model* and the *fourth-generation techniques*. Selection of one of these paradigms is dependent on the development approach to be adopted. Each paradigms possesses its own strengths and weakness and in certain instances the strongest aspects of each are combined to benefit the software project.

2.1.2.1 The Waterfall Model

The waterfall model is the most commonly known paradigm. EXECOM's APT methodology (1991), in line with other waterfall methodologies, uses a systematic, sequential approach to software development that begins at the system level and then progresses through analysis, design, coding, testing, and maintenance. This paradigm includes the following activities (Pressman, 1992, p. 25) :

(a) ***System Engineering and Analysis*** includes requirements gathering at the system level with a small amount of top-level design and analysis.

(b) ***Software Requirement Analysis*** intensifies the requirements gathering processes and focuses specifically on the software. The analyst must fully understand the information domain of the software, as well as the required functions, performance of the system and the user interface. The requirements for both the system and the software are documented and are reviewed with the customer.

(c) ***Design*** process focuses on the program's data structure, software architecture, procedural detail and interface characterisation. Before coding begins, this process translates the requirements into a form that can be assessed for quality. The design then becomes a part of the software configuration after it is documented.

(d) ***Coding*** process is where the design is translated into a machine-readable format by the programmers. Typically, a high-level programming language(s) is used to achieve this.

(e) ***Testing*** is a process of executing a program with the intention of finding error(s). It is a critical element of software quality assurance and it also represents the ultimate review of specification, design and coding. Vliet (1993, p. 12) further explains that testing is not a phase that is conducted after the implementation of the system. Testing itself can be regarded as two separate activities, namely *verification* and *validation*. Verification is to determine whether the system meets its requirement (are we building the system right). Validation is to determine whether the system meets the user's requirement (are we building the right system).

(f) ***Maintenance*** of software is something that cannot be avoided - software changes due to several reasons. The following are types of maintenance process.

Corrective maintenance is the process of removing one or more errors found on the system. *Adaptive maintenance* is the process of modifying the software to properly interface with a changing environment. *Perfective maintenance* is the process of adding or modifying of existing functions on a successful system. The final type of maintenance process is known as *preventive maintenance*. It is a process of increasing the system's future maintainability (Vliet, 1993, p. 15). Examples of preventive maintenance activities include updating of documentation, adding of comments and/or improving the modular structure of the system.

The waterfall model is probably the most common paradigm used in the software industry. The main reason for its development was that, in the past, there were not enough tools available to synthesise software (Vliet, 1993, p. 34). However, the waterfall model is considered to have a number of problems for software development. Zelkowitz (cited in Vliet, 1993, p. 34) provides sufficient quantitative evidence that the model has many shortcomings. For example, the strict sequencing of phases enforced by this model cannot always be followed.

2.1.2.2 Prototyping Model

Prototyping is a process that requires the software developer to create a preliminary model of the software to be built. **Figure 2.1.2.2.1** shows the typical prototyping approach (Alavi et al., 1991, p. 88). This model can be in three different formats (Pressman, 1992, p. 27) :

(a) a paper prototype or PC-based model that shows the human-machine interaction in a form that can be easily understood by the user.

(b) a working prototype that implements a portion of the function required by the desired system.

(c) an existing system that performs part or all the necessary function but has other features that will be improved and/or incorporated onto it.

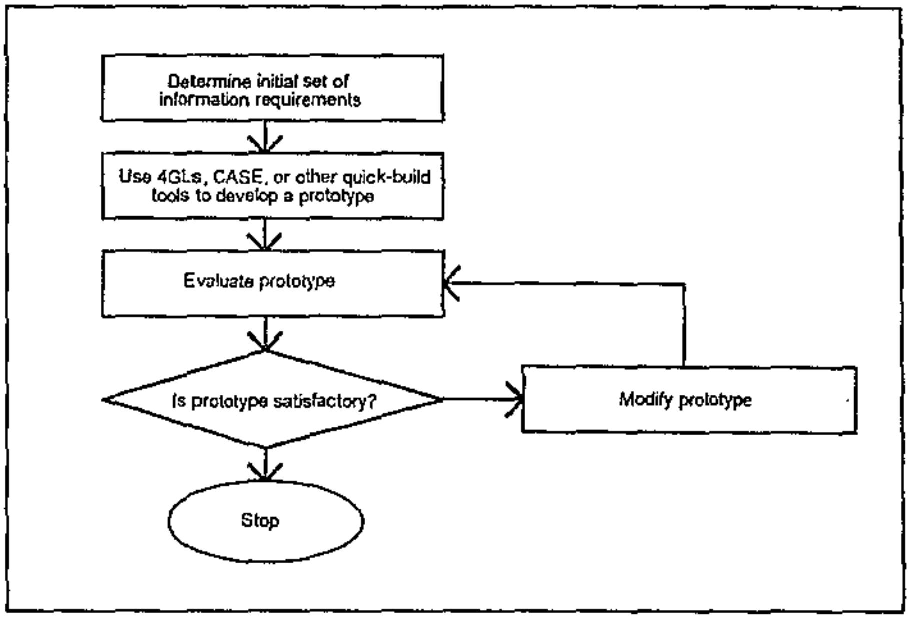

Determine initial set of information requirements
Use 4GLs, CASE, or other quick-build tools to develop a prototype
Evaluate prototype
Is prototype satisfactory?
Modify prototype
Stop

Figure 2.1.2.2.1 **Diagram - Prototyping Approach**

Prototyping is particularly useful in a situation where the users are unable to clearly define their requirements. Using prototyping, the user interface can be quickly developed, providing users with an impression of what the completed system will look like and what type of functions it will provide.

Alavi (1984, p. 562) provided four recommendations for the prototyping techniques. Alavi states that :

(a) both users and designers must be familiar with the prototyping approach and recognise its pitfalls.

(b) since prototyping is a relatively new paradigm, there is a need for a positive attitude from those who use it in order to get positive results.

(c) prototyping is very useful in situations where user requirements are unclear or ambiguous - it seems to be a good way to clarify those requirements.

(d) prototyping also needs to be planned and controlled. There must be an imposed limit on the number of iterations, and explicit procedures for documenting and testing procedures must be established. In addition, more useful aspects of the traditional paradigm that make the process manageable and controllable, should also be applied.

Alavi (1984, p. 557) conducted field interviews[3] and found the following advantages and disadvantages of prototyping. The advantages are :

- It provides a user with a tangible means of understanding and examining the proposed system and for extracting more meaningful feedback from users in terms of their needs and requirements.
- It provides a common ground where users and designers can identify potential problems and opportunities early in the development process. It also provides an effective way to extract and clarify user requirements.
- It serves as a practical means to encourage and achieve user participation and commitment to a project.
- It allows users and data processing personnel to improve communication and relationship between them, and also to enhance their appreciation of each other's job.

[3] Alavi (1984) conducted in-depth interviews with 12 project managers and 10 systems analysts from six organisations that uses the prototyping approach.

- It helps to ensure that the system will perform its expected or required tasks before spending large sums of money on the development of the entire system.

The disadvantages are (Alavi, 1984, p. 358) :

- Prototype might have limited capabilities and captures only the key features of the operational systems. Sometimes, unrealistic user expectations are created by overpraising the prototype, and these expectations are subsequently not met.
- Prototypes are difficult to manage and control, due to lack of knowledge in planning, budgeting, managing and controlling them.
- It is difficult to prototype large systems because it is unclear how a large system should be divided for the purpose of prototyping or how aspects of the system to be prototyped are distinguished and boundaries set.
- It can be difficult to retain user enthusiasm. In some cases, user involvement and interest declines after the working prototype was developed.

There are a variety of prototyping methods. Most of which aim to be more rapid than conventional development, thus reducing prototyping cost and risk (Tate, 1990, p. 240). The types of prototyping methods include (Tate, 1990, p. 240) :

- ***Ad hoc* or quick and dirty methods**
  Quick and dirty methods, in the literal sense, are often a recipe for disaster in software development. But one can assume that "quick" refers to rapid prototyping and "dirty" for the ignorance or extreme simplification of non-essentials. However, experience indicates (Tate, 1990, p. 240) that though prototypes need only be completed

in key aspects, they must be developed to a reasonable standard, especially if they are to be accepted in practice.

- ❑ **Executable specification**

One main purpose of prototyping is the determining, clarifying, or validating of user requirements. The concept of direct execution of specifications based on these requirements is very desirable. When compared with other prototyping methods, it has the great advantage of being very direct. Practically, executable specifications[4] are not quite as direct as was expected. The reason for this is because the requirements that are not explicitly specified cannot be confirmed.

If the specifications are to be executed in the normal way, they must be clear and unambiguous. This implies the use of formal specification languages[5], which unfortunately are not very user-friendly. Some research work has been conducted to develop experimental systems with "... semi-formal, graphical front-ends that are reasonably flexible and user-friendly but are supported by a more formal back-end" (Tate, 1990, p. 241).

Tate (1990, p. 241) pointed out that some might argue that executable specifications are in fact not prototyping. Specifications that can be executed are basically still specifications. Their ability to be executed is but another aspect of their understandability. Executable specifications are still extremely useful for validating requirements - which is one of the main purpose of prototyping.

[4] Executable specification is the prototype that serves as a representation of requirements (Pressman, 1992).

[5] Formal specification languages are often mathematical in form (for example, in the form of predicate calculus). It is a formal method that provides a means for specifying a system so that consistency, completeness, and correctness can be assessed in a systematic manner (Pressman, 1992).

- □ **Very High-Level Languages and Application Generators**

"Very high-level languages" refers to "... languages that are higher level or briefer and more natural in expression, than those normally used in conventional software development" (Tate, 1990, p. 241). This category includes fourth-generation languages (4GLs) or fourth-generation techniques (4GTs), various high-productivity languages that are domain specific (in varying degrees) and languages specifically developed for rapid prototyping.

All these languages and techniques have one common ability, and that is "... to specify some characteristic of software at a high level ... then automatically generate source code based on the developer's specification" (Tate, 1990, p. 241). The direct use of this high-level description on part of the system makes the use of high-level languages appropriate for rapid prototyping.

An application generator's function is very similar to that of high-level languages. It can produce a part or all of an application from suitable specifications. These specification might be expressed in graphical, tabular, menu choice or language form, or a combination of these. Some would consider application generators as a potential prototyping tool and if the code that it generates is efficient, the application generator can be considered as a high-productivity application-building tool.

- □ **Reuse**

This suggests that the prototype is assembled using a set of existing software components. A software component may be a data base, a program or a module. Each of these components can be designed in a manner that enables them to be reused without a detailed knowledge of their internal workings.

The hypothesis proposed by Alavi et al. (1991, p. 86), was that by adding data modelling as a preceding step to prototyping, it would give prototyping more structure and make it more efficient. In an experiment[6] conducted by Alavi et al. (1991, p. 86), system designers combining data modelling and prototyping, reported lower task satisfaction and more stress. It was also felt that the task was more complex. However, the experiment did confirm Alavi's hypothesis because these system designers did in fact achieve superior design results. It also showed that including the data modelling step reduces the number of prototype iterations to design the "right" system.

2.1.2.3 The Evolutionary Model

The evolutionary model is based on three simple principles (Gilb, 1988, p. 84) :

- □ *Deliver* something to a real end-user.
- □ *Measure* the added-value to the user in all critical dimensions.
- □ *Adjust* both design and objectives based on observed realities.

The basic evolutionary concepts are well-defined concepts in engineering literature and engineering practice in other disciplines. However, in the software community, its capability is yet to be fully recognised and exploited (Gilb, 1988, p. 84).

[6] The subjects for Alavi's et al. (1991) experiment were evening graduate students (52 men and 36 women) from two MIS classes at a large state university. Their average age was 26.2 and 72 percent had full or part-time professional employment in MIS.

The evolutionary model consists of a collection of many concepts. The primary concepts are (Gilb, 1988, p. 85) :

- ❑ **Multi-objective driven**

  Conventional software planning is done in terms of the functional deliverables. According to Gilb (1988, p. 86), there is very little emphasis in the industry on how quality and resource attributes of a system are controlled. As a result, control over these attributes is lost. The reason provided by Gilb (1988, p. 86), is that there is insufficient knowledge among software engineers and teachers in defining critical attributes such as usability and maintainability.

  The evolutionary model is built on iteration that leads to "... clear and measurable multi-dimensional objectives" (Gilb, 1988, p. 89). These objectives must contain all functional, quality and resource objectives that are necessary for the long-term and short-term survival of the system under development.

- ❑ **Early, frequent iteration**

  In most software engineering projects, the first useful results are delivered one or more years after the project commences. Gilb (1988, p. 89) found that the initial planners of such projects actually believe in the possibility of an earlier delivery, but they lack both motivation and method in finding early and frequent software deliveries.

  Management who desire an earlier delivery, paradoxically also believe in the conventional wisdom that there is a long initial cycle before the first useful phase is delivered. Gilb (1988, p. 89), however believes that such first phases can be sub-divided into many smaller phases, hence providing an earlier delivery.

The evolutionary planning uses the concept of selecting the most crucial steps with the highest user-value (which may be financial) to development-cost ratio for earliest implementation. This user-value might increase management goodwill and encourage their support for the rest of the system.

□ **Complete analysis, design, build and test in each step**

Software projects tend to waste a lot of time on the detailed requirements analysis, detailed design, coding and testing phases. It is a very difficult task, especially for large projects, because there are "... too many unknowns, too many dynamic changes and too many complex interrelationships in the system" (Gilb, 1998, p. 90).

The evolutionary model is created to provide developers with early warning signals of threatening unpleasant realities. Unpleasantries still exist but if they occur, they will not get a chance of becoming too large. Gilb (1988, p. 90), suggests that one must learn to design a more "open-ended" system architecture. The evolutionary model starts with an elementary design that is easy to modify, adapt, port and change - both in the long and short terms. It provides for early utilisation of the system to experience its usefulness and capabilities at an early stage.

□ **User orientation**

Software projects are mostly oriented towards the machine, the algorithm, or the deadline, but rarely towards the user. With the evolutionary model, developers are specifically appointed to "listen" to user reactions, early and frequently. The user can directly participate in the development process. In this case, neither the budget nor deadline is overrun. The overall system is "open ended" and the developers "... are mentally, economically, and technically prepared to listen to what the user or customer wants" (Gilb, 1988, p. 92).

"The principle of selecting the highest available value-to-cost ratio .... is a dynamic one" (Gilb, 1988, p.92). The user values should change as the user gains experience. This allows the user to provide new ideas that were not in the initial plans. If the idea is good, the developers must find practical and reasonable ways of implementing them as soon as possible. All developers should realise the importance of feedback, the changes of ideas about value, and the experiencing of development cost estimation.

❑ **Systems approach, not merely algorithm orientation**
Many software engineering methods are oriented towards current computer programming languages. These methods contain few references to Data Engineering aspects of software, documentation, training, marketing and motivation (Gilb, 1988, p. 93).

The evolutionary model is a method that is not merely restricted to software development. It can be used in any creative process.

❑ **Open-ended basic systems architecture**
What is most desirable from a system is one that will survive and succeed under conditions which change according to time. According to Gilb (1988, p. 93), a good software engineer should constantly be making detailed study of the available design technologies which may lead to more adaptable systems.

In terms of the evolutionary model, open architectures are vital. Without open architectures, a lot of effort will be wasted in the has an open architecture, modification or enhancement can easily be made.

- □ **Result orientation, not software development process orientation**
  In the traditional software development cycle. the process seems to be more significant than the result. Gilb (1988, p. 94) stresses that software developers are so tangled up in the formalities of a process that the software engineering efforts have "... extremely unclear, unmeasurable and unstated objectives in critical quality and resource areas" (Gilb, 1988, p. 95). It is necessary to focus on more important issues such as usability and maintainability.

Planners can choose to ignore some of these concepts, but in doing so, the model will lose some of its power.

The evolutionary model is a management perception tool. It will help management to comprehend and control the complex tasks which they are responsible for. It does this by using one of the oldest management strategies - "divide and conquer". This model breaks the task into many smaller deliverable results. The benefit of this is that the deliverable results can be used by someone trying to perform some serious work with them (Gilb, 1988, p. 112). These results have to be further adjusted, hence it does not imply a full-scale software release.

2.1.2.4 The Spiral Model

The spiral model is based on various refinements of the waterfall model. This model can accommodate the models discussed in the previous sections as special cases and also provides guidance as to which combination of the previous models best fit a given software situation.

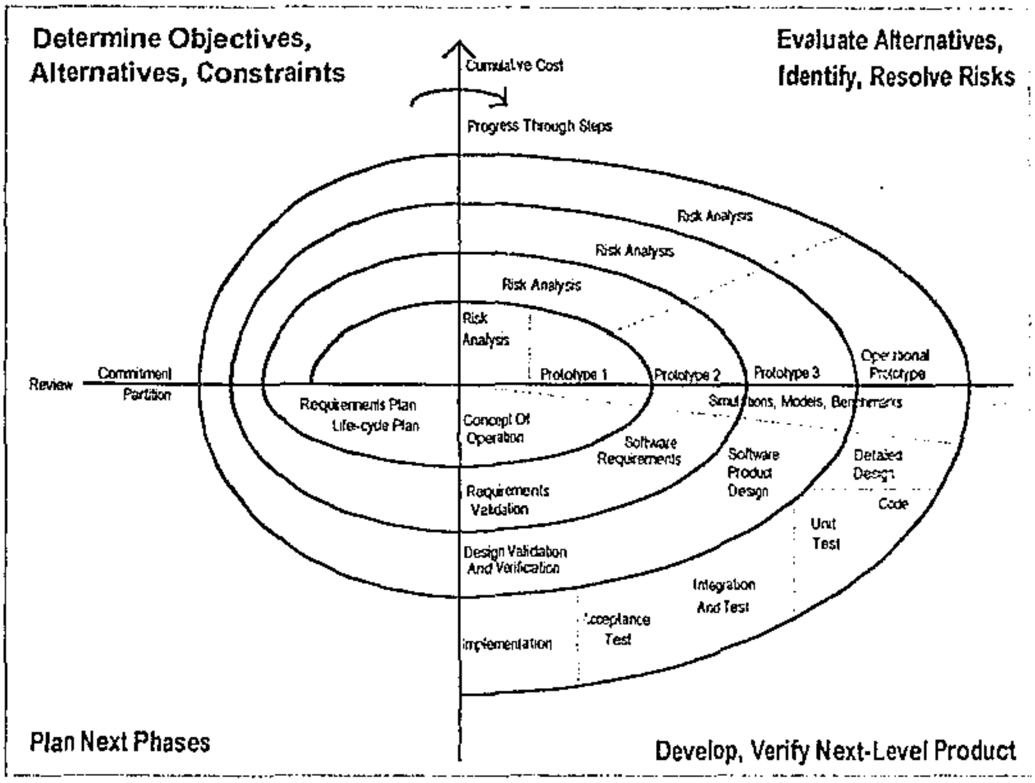


Figure 2.1.2.4.1 **Diagram - Spiral model**

Figure 2.1.2.4.1 (Boehm, 1988, p. 64) represents the spiral model of the software process. The radial dimension "... represents the cumulative cost incurred in accomplishing the steps to date" (Boehm, 1988, p. 65). The angular dimension represents the progress made in completing each cycle of the spiral. From the diagram, it can be observed that each cycle involves a advancement that addresses the same sequence of steps. Each cycle of the spiral begins with the identification of (Boehm, 1988, p. 65) :

- ❑ the key characteristics of the software such as performance, functionality, adaptability etc.
- ❑ the alternative methods of implementing the software (for example, use of design A or design B etc.).
- ❑ the constraints that are associated with the application of the alternatives such as cost, schedule etc.

The next step is to weigh the method of implementation against the key characteristics and constraints. This process usually helps to identify the areas of uncertainty that may become a risk(s) to the project. If the risk(s) is identified, the next step will be to formulate a cost-effective plan to resolve the risk. This may involve prototyping, simulation, benchmarking etc. Once the risk(s) is assessed, the next step is determined by the type of risk(s) remaining. From the next step onwards, it can be seen how the spiral model accommodates the good features of existing software development paradigms. With the risk management of the spiral model, it can avoid many of the problems that are encountered by these paradigms. For example (Boehm, 1988, p. 65) :

- □ If a project has low risk in areas such as user interface or performance, but has a high risk in budget and schedule, then the spiral model will resemble the waterfall model.
- □ If a piece of software has a low risk in design and code breakage but the presence of errors in the software constitutes a high risk, then the spiral model will resemble the two-leg model of precise specification and formal deductive program development.
- □ If a project has low risk in areas such as budget, schedule or control but has a high risk in defining the wrong user interface or user decision supports requirement, then the spiral model will resemble the evolutionary development model.
- □ If automated software generation capabilities (such as 4GL tools) are available and depending on the risk involved, the spiral model can accommodate them as an option for rapid prototyping or for application of the transform model.
- □ If the high risks found in a project involve a mix of risk items listed above, then the spiral approach will also reflect an appropriate mix of the process model.

After each cycle is completed, the software will be reviewed by the principal people or organisations concerned with it. The review involves all aspects of the software developed during the previous cycle, including the plans for the next cycle and the resources that are required to carry them out. The main objective of the review is to ensure that all parties concerned are jointly committed to the approach for the next phase. It is important to note that each component of the software can be divided to form its own spiral. Therefore, the review-and-commitment step may extent "... from an individual walkthrough of the design of a single programmer's component to a large scale requirements review involving the developer, user, customer and maintenance organisations" (Boehm, 1988, p. 65).

The spiral model has a number of additional advantages, as listed below (Boehm, 1988, p. 69) :

- ❑ It focuses early attention on the choices involving the reuse of existing software.
- ❑ It assists in the preparation for life-cycle evolution, growth, and changes of the software.
- ❑ It supplies a mechanism for combining software quality objectives into the software development.
- ❑ It concentrates on removing errors and unattractive alternatives at an early stage.
- ❑ It can deduce how much effort and resources are needed for a particular type of project.
- ❑ It does not employ different approaches for software development and software maintenance.
- ❑ It provides a practicable framework for integrated hardware-software system development.

Although the spiral model appears to be more adaptable than the other types of development paradigms, there are some difficulties that are associated with this model. Boehm (1988, p. 69) describes these difficulties as three main "... challenges that involve matching to contract software, relying on risk-assessment expertise and the need for further elaboration of spiral model steps".

❑ ***Matching to contract software.***

According to Boehm (1988, p. 70), the spiral model works well on internal software development, but it requires more work if it is to compete in the world of contract software acquisition. In the world of contract software acquisition, it is harder to procure great degrees of flexibility and freedom without losing accountability and control. It is also harder to interpret contracts whose deliverables are not well specified in advance. Although enhancement has been made to support a more flexible contract mechanism, there is still a need to ensure that the acquisition managers are comfortable in using these procedures.

❑ ***Relying on risk-assessment expertise.***

The spiral model relies heavily on the ability of the software developers to identify and manage sources of project risk. Not all software developers have the necessary experience to effectively carry out this task. For example, if a team of inexperienced developers were to produce a specification with a good level of understanding on low-risk elements but poor level of understanding on high-risk elements, the project will fail (Boehm, 1988, p. 70).

Another aspect of risk-driven specification is that they are people-dependent. For example, a design created by an expert may be implemented by non-experts. This means that the expert will have to produce very detailed documentation for the non-experts, to keep them from making mistakes.

- ❑ ***The need for further elaboration of spiral model steps.***
Basically, a lot of work has to be done on the spiral model to ensure more consistent use of the model. There is a "... need for more detailed definitions on the nature of the spiral model specifications and milestones, the nature and objectives of spiral model reviews, the techniques for estimating and synchronising schedules, and the nature of the spiral model status indicators and cost-versus-progress tracking procedures" (Boehm, 1988, p. 71). There is also a need for guidelines and checklists to identify the potential sources of project risks and their most effective risk-resolution techniques (Boehm, 1988, p. 71).

Highly experienced people will have no problems using the spiral model, but the majority of people have varying degrees of experience and understanding. Accordingly, it is important to ensure a consistent interpretation and use of the spiral approach across the project.

2.1.3 Project Management Process

It is too often the case that data processing managers struggle through huge projects, working against impossible deadlines, delivering systems that barely work and do not meet their users' requirements, and consequently later, spend a lot of time and effort on maintenance (Pressman, 1992, p. 42). This is a sign of weak project management. In order to conduct a successful software project, it is necessary to consider the following elements :

(a) **Beginning A Software Project**
Before planning a project, objectives and scope must be established, alternative solutions must be considered, and technical and management constraints must be identified. Lack of this information, makes it impossible to define an accurate estimate of the project cost, a realistic break-down of project activities, or a reasonable project schedule that provides a significant insight on progress.

(b) **Measures And Metrics**
Measurement and metrics assist in understanding the technical process that is used to develop a product and the product itself. The process is measured so that it can be improved. The product itself is also measured so that its quality can also be improved.

(c) **Estimation Process**
Estimation is an important element in managing a project. After a software project is planned, estimation is used to project the human effort required, the project duration and its cost.

(d) **Risk Analysis**
Risk analysis is another crucial element in managing a project. As stated in Gilb (1988, p. 73), "If you don't actively attack project and technical risks, they will actively attack you". Risk analysis is a series

of risk management steps that are classified as risk identification, risk assessment, risk prioritisation, risk management strategies, risk resolution and risk monitoring.

(e) **Scheduling**
After a set of project activities is identified, the interdependencies (if any) are established, the effort associated with each activity is estimated, the people and other resources are assigned, and a *task network*[7] is created. Hence, a time-line schedule is developed.

(f) **Tracking And Control**
After the development schedule is established, tracking and control activity begins. All activities on the schedule are tracked by the project manager. If any of the activities should fall behind schedule, the project manager can use a project scheduling tool to ascertain the impact of the schedule slippage on project milestones and delivery date. In doing so, the project manager can then redirect resources, reorder activities or in the worse case scenario, alter the delivery date.

2.2 SOFTWARE METRICS

Software metrics is a subject that has long been considered in the domain of software engineering. The first research work carried out was conducted by Maurice Halstead (Ince, 1990, p. 298). Halstead's study looked into the area of product metrics (see **Section 2.2.3.6**) that involves program code. The idea behind Halstead's work is that useful properties of a system or part of a system can be anticipated from counting tokens in source code. The second wave of metric research started during the 1970s. The research involved the characterisation "... of the control flow of a program or subroutine in terms of a number which, somehow, quantified its unstructuredness" (Ince, 1990, p. 298). McCabe is renowned for his

[7] A task network is a schematic on the various types of activities that are involved in the software engineering project.

study in this area. However, the most promising area of research involves system-design metrics. Such metrics can be drawn from the architectural design and measure the degree of isolation of modules in a system. It is believed that a good system is one where its modules can be read and tested in isolation, and integrated with minimum problems (Ince, 1990, p. 298).

Software metrics provide quantifiable measurement of any activity involved in software engineering. According to Fenton (1991, p. ix), such activities include matters that relate to "... measuring and predicting software project costs, measuring and improving productivity, and measuring and predicting the quality and complexity of software products". Clapp (1993) added that metrics also consist of proiect size, personnel, computer use, unit progress, schedule progress, volatility, requirements and design progress, testing progress and incremental release content.

Fenton (1991, p. ix) stresses the importance of software metrics in software engineering. He claims that even though there are literatures that talk about software metrics, they barely emphasise its importance. One main reason software engineering remains more of an ideology than a discipline is that measurement has mostly been ignored by some of the leading authorities who have shaped its direction (Fenton, 1991, p. ix). Even with books that describe methods on how to achieve software quality, many still do not know how to assess their products. Hence, it is impossible for developers to determine whether they have achieved anything even with the available methods. Many of the measuring techniques (metrics) are being used without really understanding their true purpose (Fenton, 1991, p. ix).

Software developers must recognise the principles of software metrics that involve cost, schedule and quality goals, quantitative goals, comparison of plans with actual performance throughout development, monitoring data trends for indication of likely problems, metrics presentation, and investigation of data values (Clapp, 1993). Management must balance their primary goals when selecting the metrics to use for their particular project.

2.2.1 Why Measure?

The previous section mentioned the types of activities that software metrics can be used to measure. One simple question remains : why measure? There are several reasons why a measure is necessary. According to Kizior (1993, p. 45), measures can assist a company determine whether it is competitive or not; they can assist the company to determine whether it requires improvement at its productivity and quality levels; measures can be used to assess new tools and techniques; they can help to compare results after taking some course of action and they can assist the estimating process. Ince (1990, p. 297) summarised the uses of metrics :

- □ as a means to predict the resource requirements for later parts of a software project. Since requirements are constantly changing, it is vital for developers to have the means to recalculate the project resources needed.
- □ to be used as a quality-assurance enforcement mechanism.
- □ to be used as a mechanism for assessing the performance of staff on a software project.
- □ to be used in assessing competing development methods, organisational structures and individual ways of working.
- □ to be used to assist development staff procure a quantitative estimate of the quality of their work.
- □ to be used as the foundation for intelligent and semi-intelligent software development tools.

Pressman (1992, p. 56) also said that if "... we do not measure, there is no real way of determining whether we are improving. And if we are not prevent the problems such as schedule and budget overrun, poor productivity etc. Measurement can provide benefits at the strategic level, at the project level and at the technical level. By requesting and assessing productivity and quality measures, senior management can set up important goals for improvement of the software engineering process.

2.2.2 What Are Software Metrics?

The previous two sections discussed the types of activities that software metrics can be used to measure and the reasons for measuring, but it has not explain what software metrics are. This section will explain the various categories of software metrics.

A software metric is a numerical value that is extracted from a software project. There are two types of metrics, namely, ***product metrics*** and ***process metrics*** (Ince, 1990, p. 297). Product metrics are numerical values extracted from some document, or a piece of source code. Process metrics are numerical values that depict a software process such as the amount of time require to debug a module. Metrics can also be categorised as ***result metrics*** and ***predictor metrics*** (Ince, 1990, p. 297). Predictor metrics are normally product metrics that can be used to predict the value of another metric. The predicted metric (normally a process metric) is known as a result metric (see **Figure 2.2.2.1**). Therefore, using features of a system specification to predict the amount of resources required by the software project is an example of product metrics (the system specification) being used to predict a result metrics (project resource).

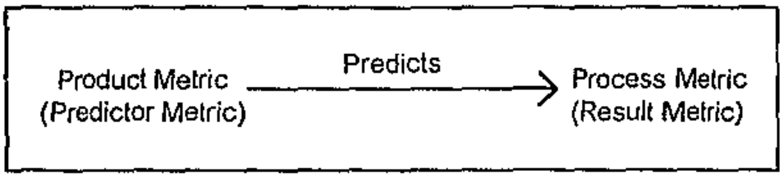


Figure 2.2.2.1 **Diagram - Relations Of Product Metric And Process Metric**

2.2.3 Types Of Software Metrics

It is now apparent that software metrics are important in software engineering. Symons (1992, p. 16) stated that "a reliable and credible method for measuring the software development cycle is needed that has a reasonable theoretical basis and that produces results that practitioners can trust." Hence, software metrics have been used to measure a wide range of software engineering activities. These activities include (Fenton, 1991, p. 9) :

- □ Cost and effort estimation models and measures
- □ Productivity measures and models
- □ Quality control and assurance
- □ Data collection
- □ Quality models and measures
- □ Reliability models
- □ Performance evaluation and models
- □ Algorithmic / computational complexity
- □ Structural and complexity metrics

For the purpose of this research, not all the metrics mentioned above will be used. For example, the cost estimation metric may not be applicable to the project that is provided by this department. According to Baker (1991, p. 1290), in order to initiate a metrics program, the following should be considered :

1. Define the object of measurement
2. Identify the attributes to be measured
3. Determine the purpose of the measurement results
4. Collect data based on steps 1, 2 and 3
5. Modify the measurement based on experience

2.2.3.1 Cost And Effort Estimation

This type of metric was first created entirely for managerial purposes. Managers wanted a method that would help them predict project costs at an early stage in the software development life-cycle. Since then, many models for software cost and effort estimation have been proposed and used. The best-known models are Boehm's ***COCOMO*** (Constructive Cost Model), Putnam's ***SLIM model*** and Albrecht's ***function point model*** (Fenton, 1991, p. 10). In these models, the general approach to estimating effort is to make effort a pre-defined function of one or more variables. These variables can be, for example, the 'size' of the software - defined as ***lines of code*** in COCOMO and number of ***function points*** in Albrecht's model.

Most cost-estimation models have adjustment factors called ***cost drivers*** built into them. These cost drivers serve as indicators for the various factors that are believed to have affect on the amount of effort required to produce a piece of software of a given size (Kitchenham, 1992, p. 212).

Boehm's COCOMO

Boehm introduces a hierarchy of software estimation models (*COCOMO*) that takes three forms. They are :

- □ **Basic COCOMO**

  This model is applicable to small-to-medium size systems usually developed in an in-house environment. Other aspects of this model includes phase distribution of effort, schedule and activities. It is suitable for quick, early rough estimation of software costs, but its accuracy is rather restricted because it lacks in factors such as hardware constraints, personnel quality and experience, use of modern tools and techniques, and other factors that might have significant impact on software costs (Boehm, 1981, p. 58).

- □ **Intermediate COCOMO**
  This model is a compatible extension of the Basic COCOMO model. It has greater accuracy and is more detailed. This makes it more suitable for cost estimation at the more detailed stages of software product definition (Boehm, 1981, p. 114). It also embodies an additional 15 predictor variables known as ***cost drivers***. These cost drivers are further explained later in this section. However, this model has two limitations which affects detailed cost estimates for large software projects. These limitations are (Boehm, 1981, p. 344) :

  - Its estimated distribution of effort by phase may be inaccurate.
  - It can be unmanageable to use on a product with many components.

- □ **Advanced COCOMO**
  This model addresses the limitations found in Intermediate COCOMO. It overcomes these limitations by providing (Boehm, 1981, p. 344) :

  - a set of ***Phase-Sensitive Effort Multipliers*** for each cost driver attributes. By using these multipliers, the amount of effort required to complete each phase can be determined.

- a ***Three-Level Product Hierarchy***, where the same cost drivers may be applied to components that are grouped at module, subsystem or system level.

This model includes capabilities such as a procedure for adjusting the phase distribution of the development schedule. For estimating overall development schedule and effort distribution by activities, this model uses the same techniques used in Intermediate and Basic COCOMO.

COCOMO can be applied to three classes of software projects, which Boehm calls ***organic mode, semi-detached mode*** and ***embedded mode*** (Vliet, 1993, p. 103). Organic mode refers to relatively small, simple software projects that involve small project teams whose members generally have lots of experience with similar projects in their organisation. Semi-detached mode refers to intermediate software projects whose project members consist of mixed levels of experiences (including those that have no experience at all). Embedded mode refers to software projects that must be developed within a set of tight hardware, software and operational constraints.

COCOMO model is associated with a set of 15 cost driver attributes that are grouped into four categories, namely product attributes, hardware attributes, personnel attributes and project attributes. Each of these 15 attributes is associated with a rating of 1 to 6 points, 1 being "very low" and 6 being "extra high". Based on these ratings, the ***effort multiplier*** can be determined from a table published by Boehm, and the product of all the effort multipliers will give the ***effort adjustment factor***.

Product attributes include :

□ **Required software reliability**
A software can be said to be reliable if it can perform its intended tasks satisfactorily. Quantitatively, software reliability can be defined as a probability. An unbiased estimator (**R**) can be obtained for the probability by performing the following steps (Brown & Lipow cited in Boehm, 1981, p. 372) :

- Choose N inputs or input sequence randomly from the operational profile distribution
- Use the inputs to exercise the software for **N** runs
- Use the success criterion to determine how many runs resulted in satisfactory outcomes (**M**).
- Calculate the estimator **R = M / N**

□ **Size of application database**
The amount of effort required to develop a piece of software depends on the size and complexity of the data base. It is very difficult to characterise the specific attributes of the software data base which influence the software's cost. Most software complexity metrics have concentrated on program complexity and exclude data complexity. The size of the data base (**D/P**) can be defined as a ratio of (Boehm, 1981, p. 386)

$$D / P = \frac{\text{Data base size in bytes or characters}}{\text{Program size in number of delivered source instructions}}$$

where data base size refers to the amount of data to be assembled in storage by the time of software acceptance.

- ❑ **Complexity of the product**
  In this case, the effort multiplier is presented as a function of the level of complexity of the module to be developed. A rating is given to the function operated by the module. These functions can be control, computation, device-dependent, or data management operations (Boehm, 1981, p. 390).

Hardware attributes include :

- ❑ **Run-time performance constraints**
  The effort multiplier is presented as a function of the degree of execution time constraints imposed on a software subsystem. "The rating is expressed in terms of the percentage of available execution time expected to be used by the subsystem and any other subsystems consuming the execution time resource" (Boehm, 1981, p. 401).

- ❑ **Memory constraints**
  The effort multiplier is presented "... as a function of the degree of main storage constraint imposed on a software subsystem. Main storage refers to direct random access storage such as core, integrated-circuit etc., but excludes devices such as drums, disks, tapes, or bubble storage" (Boehm, 1981, p. 410).

- ❑ **Volatility of the virtual machine environment**
  The effort multiplier is presented as a function of the level of volatility of the virtual machine based on the subsystem to be developed. In a given software subsystem, the underlying virtual machine is a composite of hardware and software that the subsystem calls upon to achieve its tasks (Boehm, 1981, p. 413).

❑ **Required turnaround time**
The effort multiplier is presented as a function of the level of computer response time experienced by the project team developing the subsystem. It is defined in terms of average response time measured in hours (Boehm, 1981, p. 415).

Personnel attributes include :

❑ **Analyst capability**
A different rating is given to the level of capability of the analysts working on the software subsystem. For each rating, a set of multipliers is to be multiplied to account for the difference in the capability of the analysts (Boehm, 1981, p. 427).

❑ **Programmer capability**
The effort multiplier is presented as a function of the level of capability of the programmers working on the software module. The ratings are represented in terms of percentiles (Boehm, 1981, p. 435). The major factors that are considered include :

- Programmer's ability
- Efficiency and thoroughness
- Ability to communicate and cooperate

❑ **Applications experience**
The effort multiplier is presented as a function of the level of applications experience of the project team. The ratings are defined in terms of experience in a particular type of application (Boehm, 1981, p. 431).

❑ **Virtual machine experience**
The effoit multiplier is presented as a function of the level of virtual machine experience of the project team (Boehm, 1981, P. 439).

❑ **Programming language experience**
The effort multiplier is presented as a function of the level of programming language experience of the project team. The ratings are defined in terms of experience with the programming language used (Boehm, 1981, p. 442).

Project attributes includes :

❑ **Use of modern programming practices**
The effort multiplier is presented as a function of the degree to which modern programming practices are used (Boehm, 1981, p. 451). Such practices includes :

- Top-down requirements analysis and design
- Structured design notation
- Top-down incremental development
- Design and code walkthroughs or inspections
- Structured code
- Program librarian

❑ **Use of software tools**
The effort multiplier is presented as a function of the degree to which software tools are used (Boehm, 1981, p. 459).

❑ **Development schedule constraint**
The effort multiplier is presented as a function of the level of schedule constraint imposed on the project team. The ratings are defined in terms of the percentage of schedule stretch-outs or acceleration (Boehm, 1981, p. 466).

Even though COCOMO is well-known and widely used, there are still some criticisms about it's approach, as Kitchenham (1992, p. 213) pointed out. First, the COCOMO model has 15 cost drivers and many are treated as if they are independent of one another, but there is evidence that they are not. A report produced by Kitchenham (1992, p. 214) states that project teams with high virtual machine experience usually have high programming-language experience, hence there is a relationship between the two factors. Secondly, the model assumes that the factors are applicable in all organisations and thirdly, the factors require a subjective evaluation. This is a problem because it is very difficult to ensure that different estimators make subjective assessments in the way as described by the model's builder.

Putnam's SLIM Estimating Model

The *SLIM estimating model* was developed by Larry Putnam of Quantitative Software Management in the late 1970s (Kemerer, 1987, p. 417). Putnam's *SLIM model* "is a dynamic multivariable model that assumes a specific distribution of effort over the life of a software development project. The model was derived from labour distributions encountered on large projects" (Pressman, 1992, p. 87). The distribution effort is presented graphically by what is known as the ***Rayleigh-Norden curve*** (Figure 2.2.3.1.1).

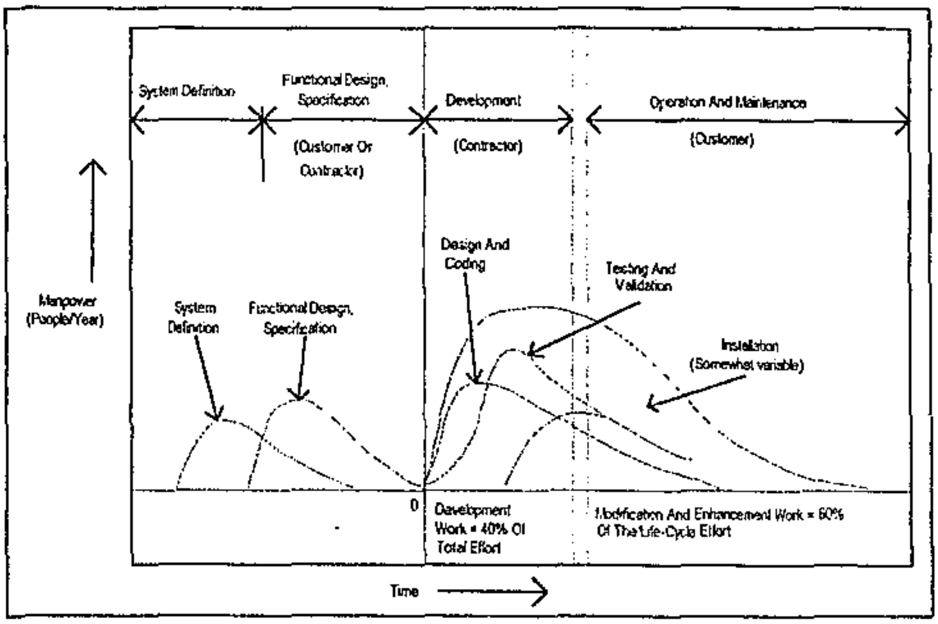

System Definition
Functional Design, Specification
(Customer Or Contractor)
Development
(Contractor)
Operation And Maintenance
(Customer)
Manpower (People/Year)
System Definition
Functional Design, Specification
Design And Coding
Testing And Validation
Installation (Somewhat variable)
0
Development Work = 40% Of Total Effort
Modification And Enhancement Work = 60% Of The Life-Cycle Effort
Time

Figure 2.2.3.1.1 **Diagram - Putnam's SLIM Model**

The Rayleigh-Norden curve can be used to derive the "software equation" that relates the number of delivered lines of code (***L***) to effort (***K***) and development time (***t***). The software equation is (Pressman, 1992, p. 87) :

$$L = C_k K^{1/3} t_d^{4/3}$$

where C_k is a state-of-technology constant and reflects the throughput constraints that affect the progress of the programmer. For example, if C_k = 2000, that suggests a poor software development environment (such as no methodology or poor documentation). If C_k = 8000 or 11000, that suggests a good or excellent software development environment, respectively. The constant C_k can be derived for local conditions using historical data collected from past development efforts.

The equation above can be rearranged to form the expression for development effort (***K***). The expression for development effort is as follows (Pressman, 1992, p. 88) :

$$K = \frac{L^3}{C_k^3 t_d^4}$$

where K is effort expended (in person-years) over the entire life cycle for software development and maintenance, and t_d is the development time in years. This equation can be related to development cost by including the labour rate factor ($/person-year).

In a study conducted by Kemerer (1987, p. 420), the SLIM model was used to estimate software costs based on the data gathered from 15 large completed business data-processing projects. From the study, it is shown that the SLIM model does not do well via the ***magnitude of relative error***[8] (or MRE) test. MRE is defined as :

$$\text{MRE} = \frac{\text{MM}_{\text{est}} - \text{MM}_{\text{act}}}{\text{MM}_{\text{act}}}$$

where MM_{est} is the estimated effort and MM_{act} is the actual effort.

The average percentage error is 772 percent, with the smallest error being 21 percent. It also shows that the errors are all biased and effort is overestimated in all 15 cases. Kemerer (1987, p. 422) suggested that this may be due to the fact that SLIM was originally developed using data from defence-related projects where productivity is usually lower than those business data-processing systems.

Albrecht's Function Point Analysis

Function point analysis is a technique that helps programmers to estimate efficiently the amount of time required to develop an application, based on its complexity (Davis, 1992, p. 88). This estimation method increases the effectiveness of project management as developers have a better idea how to schedule programming time and allocate resources. Davis (1992, p. 88) also added that estimation based on this method can vary by as much as ± 35 per cent during the early stages of the development cycle and by as little as 10 per cent during design definition stages. More of function point analysis is discussed in **Section 2.4**.

[8] The MRE test is used to determine the errors of underestimating and overestimating the amount of effort put into the projects.

Lines Of Code Method

One of the main criticisms concerning function points is that they are subjective whereas lines of code are objective. Counting function points still requires human involvement, and this implies subjectivity. However, it is not entirely true that lines of code are an objective metric (Jones, 1991, p. 49). There are three problems associated with lines of code.

- □ There are no national or international standard for a line of code that encompasses all procedural languages. Ever since the inception of the software industry, lines of code have been used. According to Jones (1991), it is very surprising that after all this time, the basic concept of a line of code has never been standardised.

- □ Currently, software can be produced using methods such as application generators, spreadsheets, graphic icons, reusable modules of unknown size and inheritance. For software developed using either of these methods, entities such as lines of code are totally inapplicable.

- □ The number of lines delivered will be less as the level of language gets higher. So, the most powerful and advanced languages will appear less productive than the more primitive low-level languages.

Software cost estimation models serve as an essential foundation for software project planning and control. Only when a software project has clear definitions of its primary milestones and reasonable estimates of the time and money it will require to accomplish them, a project manager cannot tell whether his/her project is under control (Boehm, 1984, p. 19).

However, according to Kusters et al. (1990, p. 190), after evaluating a number of selected cost estimation models[9], they concluded that these models cannot accurately measure software cost. The models need to be adapted into the environment in which they will be used. In Kemerer's (1987, p. 427) paper, the conclusion that was derived was that models that were developed on different environments do not work well uncaliberated, hence calibration is essential. Kusters et al. (1990, p. 190) also added that, despite the great number of publications on cost estimation models, they were unable to find any empirical data that shows the capability of these models to predict effort and software cost accurately. They believed that an organisation should not completely relies on the estimates derived from a single model.

2.2.3.2 Productivity Measures And Models

Almost everyone with experience of working in large software projects, knows that by putting more people on to a late project will delay the project even more (Brooks cited in Fenton, 1991, p. 260). Productivity metrics are used to measure the productivity of personnel during different software processes and in different environments. The model shown in Figure 2.2.3.2.1 (Fenton, 1991, p. 11) identifies that productivity is a function of ***value*** and ***cost***. It endeavours to determine the individual components of these in measurable form. Fenton also suggested the productivity model will project a more accurate view of productivity than models measuring *size* of output divided by *effort*.

Fenton (1991, p. 262) pointed out that in general, people do not like to be monitored and measured. If people know that they are being evaluated, there is a temptation by them to manipulate the data. Hence, he

[9] Kusters et al. (1990) selected Before You Leap, Estimacs, SPQR20 and BIS/Estimator as the cost estimation models for their study.

suggests that productivity should be viewed as an attribute of the human resource. After all, the measuring of productivity can be viewed as the measuring of a major software resource ... people! In this context, productivity refers to the people working on a part(s) of the development of the software such as coding, documenting etc. Therefore, productivity can be viewed as an external resource attribute.

According to Horst Remus (cited in Gilb, 1988, p. 256) of IBM, productivity improvement techniques must be focused more on management than on software developers. Gilb (1988, p. 257) himself added that many software developers believed that productivity can be improved by using more sophisticated programming langu<span>ages and/or more sophisticated software support tools. There is some truth in this viewpoint but as Remus concluded from his observation at IBM (cited in Gilb, 1988, p. 256), productivity will greatly improve if the productivity of management is improved - not through technical means.

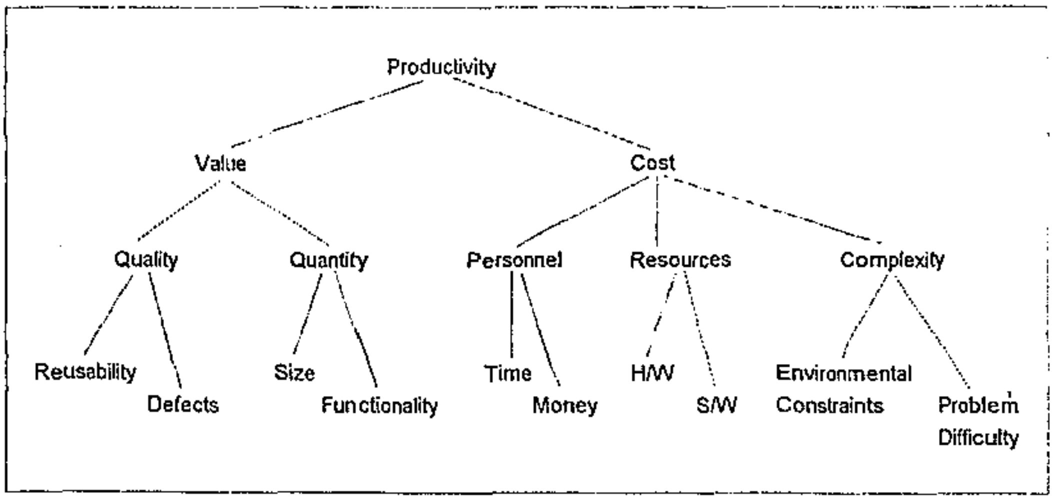


Figure 2.2.3.2.1 **Diagram - Fenton's Productivity Model**

Even though there are many problems associated with measuring productivity using the line-of-code (LOC) approach, many companies will continue to use this method simply because it can be relatively easy to compute automatically (Fenton, 1991, p. 265). However, there is another approach that may prove to be a better productivity measure ... the function point method.

Behrens (1983, p. 649) did a study to determine the productivity of application development using the function point method. Behrens collected data from 11 projects completed in 1980 and 14 projects completed in 1981. The function point data were collected manually on a specially designed form. These data went through extensive review to ensure consistency and uniformity. Cost data were collected from an automated project management system. Consulting and user time was collected manually from the project records. The time data went through extensive auditing to ensure accuracy.

From the study, by mapping the project cost against the project size (function points), it shows that if the project size increases, their unit costs also increases. Behrens (1983, p. 649) states that this is a significant productivity result. The same result was derived when productivity (hour/function point) is mapped against project size.

Behrens then examined two major attributes of these projects : development environment and programming language. The results showed an average unit cost of 0.77 for the on-line environment and 1.52 for batch (Behrens, 1983, p. 650). This is the second important productivity result because it shows that the productivity for the on-line environment is approximately twice that of batch.

The languages that were used include Wang Utilities, Databus, Focus, CMS Exec, PL/1 and COBOL. From the study, it shows that Wang Utilities is 41 per cent less costly than Focus and 67 per cent less costly than COBOL (Behrens, 1983, p. 651). Behrens pointed out that language is dependent of the development environment.

The final analysis showed that project size, development environment, and language are determinants of system development productivity. Other project attributes such as years-of-systems-experience and user experience (customer's people working on the project) were also tested but found not to be significant in Behrens' study. Although the data from Behrens' study is old, it does show that the function point method can be used as a general measure of development productivity.

2.2.3.3 Quality Models And Measures

Most experts believe that even with metrics that can accurately estimate software cost and measure productivity, it will not guarantee the success of the software if ***quality*** is not considered. ***Total Quality Management*** (TQM) was introduced to the software world from industry, where it had proved very effective in ensuring the quality of the finished product. Keyes (1992) stated that "... TQM focuses on the product and is a process whereby continuous improvement is constantly stressed". It is also added that many ***Information Systems*** (IS) only use TQM in the early stages of software development. Fewer than 5 per cent of these organisations maintain the quality improvement process throughout the product life cycle. Management must realise that if TQM is not enforced at an early stage, the cost of detecting and repairing of defects, and software maintenance will be high.

McCall's model and Boehm's COCOMO model are two well-known software quality models. McCall's and Boehm's models attempt to identify key attributes of quality from the user view of the final product.

These attributes are usually called ***quality factors*** (Vliet, 1993, p. 71). McCall described these quality factors as (Pressman, 1992, p. 551) :

- ❑ ***Correctness*** : The degree to which the program satisfies the user's requirement.
- ❑ ***Reliability*** : The degree to which the program is expected to perform its intended function with acceptable precision.
- ❑ ***Efficiency*** : The amount of computing resources and code required by the program to perform a task.
- ❑ ***Integrity*** : The degree to which access to the software or data by unauthorised persons can be controlled.
- ❑ ***Usability*** : The effort required to learn, operate, prepare input, and interpret the output of the program.
- ❑ ***Maintainability*** : Generally, the effort required to locate and fix an error in a program.
- ❑ ***Flexibility*** : The effort required to modify a working program.
- ❑ ***Testability*** : The effort required to test a program to ensure that it is performing its intended function.
- ❑ ***Portability*** : The effort required to transfer a program from one hardware and/or software system environment to another.
- ❑ ***Reusability*** : The degree of a program or part of a program, that can be reused in other applications.
- ❑ ***Interoperability*** : The effort required to link one system to another.

These attributes are often considered ***too*** high-level to be meaningful and measurable directly. Hence, these high-level attributes are decomposed into lower-level attributes called ***quality criteria*** (Fenton, 1991, p. 223). The quality criteria again require one further level of decomposition to associate them with a set of low-level, directly measurable attributes known as ***quality metrics*** (Fenton, 1991, p. 225).

There are two types of attributes namely, internal and external attributes. According to Vliet (1993, p. 71), internal attributes of a piece of software can be measured purely in terms of the software itself. Examples of internal attributes are modularity, size, defects encountered etc. External attributes of the software can only be measured with respect to how the software relates to its environment. Examples of external attributes are maintainability, usability, reliability etc. In many cases, the quality criteria of the internal attributes may have direct impact on the external quality attributes. For example, the reliability (external attribute) of the software cannot be directly measured. To measure reliability, it is necessary to directly measure the number of defects (internal attribute) encountered on the software so far. This direct measure can provide an insight to the reliability of the software.

The idea of quality on a piece of software varies from person to person. This is true in the case of software quality. The software engineers, project management and the client may have different definitions of what quality is. There might also be trade-offs between the various quality attributes such as maintainability and timeliness (Shepperd, 1990, p. 312).

2.2.3.4 Reliability Models

Musa and his colleagues (cited in Pressman, 1987, p. 459) describe software reliability models in the following manner : "Software reliability models are used to characterise and predict behaviour important to managers and engineers. In order to model software reliability one must first consider the principle factors affecting it : fault generation, fault removal and the environment. Fault generation depends primarily on the characteristics of the developed code (code created or modified for the application) such as size and development process characteristics such as software engineering technologies and tools used, level of experience of personnel, etc. Note that code can be developed to add features or to remove faults. Fault removal depends on time, operational profile, and the quality of the repair activity. The environment depends on the operational over time, software reliability models are generally formulated in terms of random processes".

There are two categories of a software reliability model. One model predicts reliability as a function of chronological (calendar) time. The other model predicts reliability as a function of elapsed processing time (CPU execution time). According to Musa and his colleagues, the model based on CPU execution time reveals the best overall results (Pressman, 1992, p. 583). There are two models, based on CPU execution time, which are not too complicated and yet yield fairly good results. They are the *basic execution model* and the *logarithmic Poisson execution time model* (Vliet, 1993, p. 360).

With all these reliability models around, it is difficult to conclude that there is one measuring technique that can consistently give accurate results over different data sources. So in practice, what developers have done is to use several measuring techniques in a particular case, hoping to select one (if any) that will produce the more trustworthy results.

2.2.3.5 Performance Evaluation And Models

This involves the measurement of a specific software product attribute ... *efficiency*. Evaluation of performance includes external system performance aspects such as ***response times*** and ***completion rates***. It also evaluates the performance of internal workings of a system such as the ***efficiency*** of algorithm (Fenton, 1991, p. 13).

Systems performance evaluation has been developed mainly in isolation with respect to other disciplines such as computer architecture, system organisation, operating systems, and software engineering (Ferrari, 1986, p. 678). Ferrari proposed several answers for the cause of this compare to other scientific fields, and it is one that is still rapidly developing. Ferrari (1986, p. 679) suggests that perhaps it is this rapid advancement that has "... characterised this field so far, there has been little incentive for reflection, and the quantitative evaluation of system performance certainly requires a more reflexive attitude than the introduction of new, more powerful functionalities".

Another likely reason for the isolation of performance evaluation is that computers are very complex machineries. This is because, it is extremely hard to quantify the needs and the behaviour of their human users. The third likely reason as proposed by Ferrari (1986, p. 679), suggests that a sizeable fraction of computer scientists view the field of computer science as an art form, thus cannot and should not be subjected to quantitative assessment.

2.2.3.6 Structural And Complexity Metrics

Structural complexity metrics are mainly used for measuring specific quality attributes such as ***reliability*** and ***maintainability***. However, these attributes cannot be measured until some working model of the code is available. From the developers' point of view, it is desirable to be able to predict which parts of the software system are likely to be less reliable or require more maintenance than others. The type of metrics used are McCabe's cyclomatic and Halstead's complexity metrics (Pressman, 1992, p. 573).

McCabe's Complexity Model

To determine the complexity of a software, McCabe suggests a "mathematical technique that will provide a quantitative basis for modularisation and allow us to identify software modules that will be difficult to test or maintain" (Shepperd, 1988, p. 30). He suggested that the number of control paths through a module would be a better indicator, since this is distinctly related to testing effort. McCabe's model uses classical graph theory to describe the complexity of the software. This method counts the number of edges in the program (*e*), the number of nodes (*n*), and the number of connected components (*p*). Hence, the cyclomatic number of the program can be calculated using the formula (Shepperd, 1988, p. 31) :

$V(G) = e - n + 1$ (**See Figure 2.2.3.6.1**)

where *V* is the cyclomatic complexity and *G* as the program graph. In the case, where there are more than one component, the cyclomatic complexity can be calculated using the formula (Shepperd, 1988, p. 31) :

$$V(S) = e - n + 2p$$

where *S* is a set of connected components. Each component must contain a single entry and a single exit node.

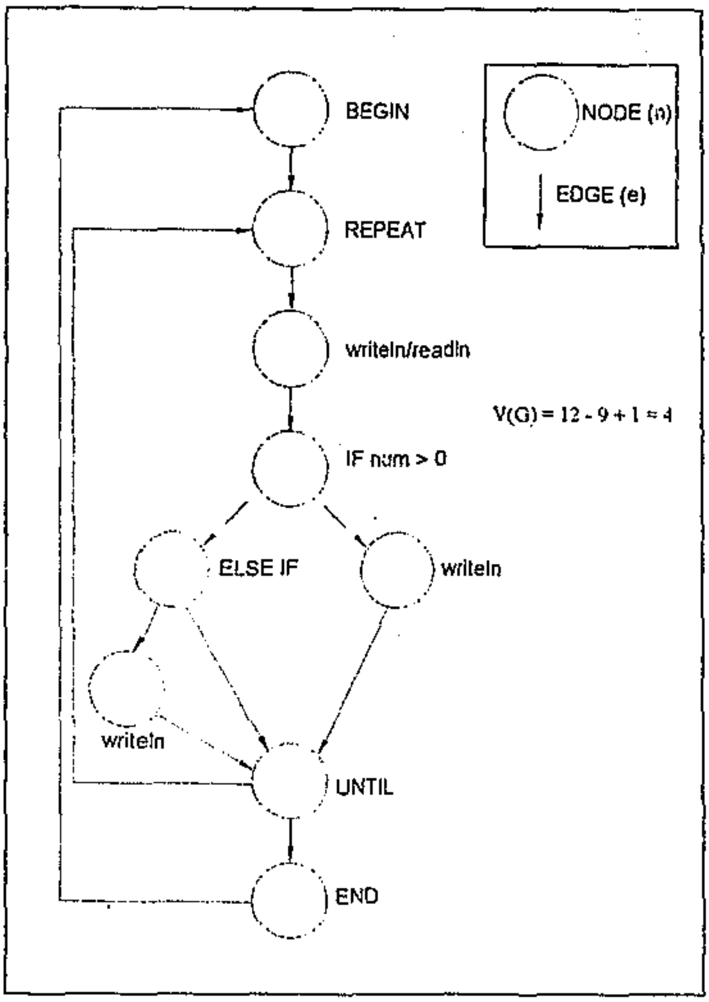


Figure 2.2.3.6.1 **Diagram - Derivation of *V(G)* for an example program**

Halstead's Software Science

Software science was introduced by Maurice H. Halstead. Its main concern was with the implementation of algorithms as computer programs (Felican et al., 1989, p. 1630). Halstead's theory of software science is possibly the best known and most thoroughly studied (Pressman, 1992, p. 573). Software science uses a set of primitive measures that may be derived after code is generated or estimated once design is complete. These primitive measures are (Curtis et al., 1979, p. 98) :

- ❑ n_1 - the number of distinct operators that appear in a program
- ❑ n_2 - the number of distinct operands that appear in a program
- ❑ N_1 - the total number of operator occurrences
- ❑ N_2 - the total number of operand occurrences

The ***operators*** can be regarded as the language's standard operators (for example, "+", "-", "*" etc) and ***keywords*** (such as IF-THEN-ELSE, BEGIN-END statement etc) and the ***operands*** can be regarded as the ***variables*** and ***constants*** used by the programmer (Felican et al., 1989, p. 1630). From these primitive measures, Halstead was able to develop expressions for :

- ❑ the overall program ***length*** (*N*)
- ❑ the potential minimum ***volume*** (*V*) for an algorithm
- ❑ the actual volume (the number of bits required by a program)
- ❑ the ***program level*** (a measure of software complexity)
- ❑ the ***language level*** (a constant for a given language)
- ❑ development effort (*E*)
- ❑ development time (*T*)
- ❑ the projected number of faults in the software.

Halstead shows that the length *N* can be estimated using the equation (Pressman, 1992, p. 573) :

$$N = n_1 \log_2 n_1 + n_2 \log_2 n_2$$

and program volume *V* may be defined as :

$$V = N \log_2 (n_1 + n_2)$$

However, it should be noted that *V* may vary depending on the programming language used and the volume of information (in bits) required to a specific program.

In theory, a minimum volume must exist for a particular algorithm. Halstead defines a volume ratio ***L*** as the ratio of the volume of the most compact form of a program to the volume of the actual program. In actuality, ***L*** must always be less than 1. Using the primitive measures, the volume ratio may be expressed as (Pressman, 1992, p. 575) :

$$L = \left(\frac{2}{n_1}\right) * \left(\frac{n_2}{N_2}\right)$$

Halstead proposed that each language be categorised by a language level (*l*), which varies among languages. He theorised that *l* is a constant for a given language, but other work indicates that *l* is a function of both the language and the programmer (Pressman, 1992, p. 575).

The effort (*E*) required to develop the software can be approximated by the equation (Mills, 1988, p. 12) :

$$E = \frac{n_1 n_2 [n_1 \log_2 n_1 \ + \ n_2 \log_2 n_2] \log_2 n}{2n_2}$$

where *n* can be obtained from the relationship

$$N = n \log_2 (n/2)$$

The corresponding programming time (*T*, in seconds) can be derived from ***E*** by dividing by the Stroud number (*S*). The Stroud number is usually taken as 18 for these calculations (Mills, 1988, p. 12).

$$T = \frac{E}{S}$$

However, if only the value of length (*N*) is known, then time (*T*) can be approximated using this equation (Mills, 1988, p. 12) :

$$T = \frac{N^2 \log_2 n}{4S}$$

Halstead's theory has generated some controversy and not everyone agrees that the underlying theory is correct. But experimental verification of Halstead's findings have been conducted for a number of programming languages. In particular, Felican et al. (1989, p. 1630) conducted an experiment by examining about 550 Pascal programs in the data processing centre of the University of Udine, which represent the widest test of Halstead's theory with regard to Pascal programs. They concluded that Halstead's formulas underestimate the number of total operators for programs written in high level languages such as Pascal. They suggested that the reason for this inconsistency was derived from the nature of the language itself.

2.2.4 Data Collection

It would be ideal to be able to gain control over the software process by accurately predicting and measuring software cost and personnel productivity. However, this all depends on how careful and well planned the task of collecting data is carried out. Even with the "best" metric around, if the data collection method is poor and inconsistent, the results derived from the metric would be rendered meaningless. The collection of data requires human observation and reporting. This requires managers, system analysts, programmers, testers and users to record *raw data* on forms.

Manual recording of data is associated with problems such as bias, error, omission and/or delays. Therefore, ***automatic*** data capture is more desirable. However, to ensure the accuracy and completeness of data, much human intervention is required. Hence, in most cases, the manual recording technique is still the best.

Basili et al. (1984, p. 728) suggest the use of a goal-directed data collection method. This model starts with a set of goals that are to be satisfied. These goals are used to generate a set of questions that are to be answered. It then proceeds step-by-step through the design and implementation of a data collection and validation mechanism. Analysing the data may provide answers to the questions and it may also generate a new set of questions. This model relies heavily on an interactive data validation process - the people who supply the data are interviewed for validation purposes concurrently with the software development process (Basili et al., 1984, p. 728). The model that Basili et al. (1984, p. 729) proposed consists of six basic steps, with considerable feedback and iteration occurring at several different places. These steps are :

- □ **Establish the Goals of the Data Collection**
  According to Basili et al. (1984, p. 729), the goals that are set, reflect the type of development methodology used. A goal is to assist in the understanding of the environment and to focus on the attention of techniques that are applicable in that environment. Without a goal, the data collected might end up being incomplete or irrelevant. *Example of a goal - to add new piece of functionality to an existing system.*

- □ **Develop a List of Questions of Interest**
  After the goal(s) has been conceived, it can be used to develop a list of questions that are to be answered. Without these questions, data distributions that are needed for assessment purposes may have to be produced in an ad hoc manner, and be incomplete or inaccurate. *Example of a question of interest might be - "What is the distribution of changes across system components?"*

- □ **Establish Data Categories**

  After the questions of interest have been set up, a categorisation scheme must be created. Each categorisation scheme must be complete and consistent. Each category can be further sub-categorised. *Example of main data category - Modification. Example of sub-category for Modification can be "optimise system performance", "change development support environment" etc.*

- □ **Design and Test Data Collection Form**

  A data collection form is used to provide a permanent copy of the data and to reinforce the programmers' memories. Designing forms can be a very tricky process because they often represent a compromise among conflicting objectives. The form must be designed so that the data collected can be used to answer the questions of interest.

- □ **Collect and Validate Data**

  Once the forms have been filled in by the necessary people, they are checked for correctness, consistency and completeness. During the validation process, if the checks reveal some problems, the people who filled in the forms will be interviewed.

- □ **Analyse Data**

  The data are analysed by calculating the parameters and distributions needed to answer the questions of interest.

2.2.5 Future Directions Of Software Metrics

The history of software metrics has been dominated by product metrics. Furthermore, these metrics have been applied only to conventional notations used in the development of software using procedural languages. According to Ince (1990, p. 300), there is a need for more research on software metrics in other areas. For example :

- ❑ There is a need for data metrics to measure the unstructuredness of the stored data in an application.
- ❑ More study is needed on the use of mathematical notations for specification and system design that are often used in safety-critical systems.
- ❑ Not much is known about metrics that can be performed on non-procedural languages like Prolog, because there is no notion of control flow.
- ❑ Not much work has been conducted on notations for specification. Most of the research is concentrated on resource estimation. There is a major requirement for research into metrics for the maintenance of such notations.

2.3 CAPABILITY MATURITY MODEL

For the past two decades, new software methodologies and technologies have not resulted in a significant improvement in software productivity and quality. Both industry and government organisations have realised that the problem lies in their inability to manage the software process. Even with the best methods and tools, developers cannot possibly hope to achieve their goals when the project is disorganised.

2.3.1 Immature Versus Mature Software Organisations

Organisations are require to understand the differences between immature and mature software organisation before they can set any goals for process improvement. An immature software organisation is one where the software processes are defined by developers and management during the duration of the project. Based on unrealistic estimates, the project schedules and budgets are often inaccurately projected. In situation where the project is behind schedule, product functionality and quality are often compromised, and activities such as reviews and testing are frequently eliminated (Paulk et al., 1993, p. 2). There will be no means of judging the product's quality or for solving problems associated with the product or process objectively. This makes it very difficult to predict the quality of the product.

A mature software organisation is an organisation that has full control over the software development and maintenance processes. Members of the development staff are fully aware of the software process and the work activities are executed as planned. The roles and responsibilities for each process are carefully defined and are made clear throughout the entire organisation and the project. Software quality and customer satisfaction are monitored by the managers. Unlike the immature software organisation, product quality is objectively and quantitatively measured. The problems that are associated with product and process are carefully analysed. Project schedules and budgets can be realistically estimated, based on historical data. By doing so, the proposed development cost, schedule, software functionality and quality of the software are usually realised (Paulk et al., 1993, p. 2)

It is obvious that there is a need for a software process maturity framework. This framework serves as a evolving path from ad hoc, undiscipline processes to mature, disciplined software processes. This framework acts as a foundation where initial improvement programs can be established. Having established the initial foundation, future improvement programs can be further applied onto the framework. The software process maturity framework is created based on the combined concepts of software process, software process capability, software process performance and software process maturity (Paulk et al., 1993, p. 3).

2.3.1.1 Software Process

According to Paulk et al. (1993, p.3), a *software process* can be defined as "... a set of activities, methods, practices and transformations ..." that developers use in the development and maintenance of software and other associated products such as project plans, design documents and code. As the organisation matures, the software process also matures and will be more consistently implemented throughout the organisation.

2.3.1.2 Software Process Capability

Software process capability describes the results that can be accomplished after following a software process. It provides the organisation with a means of predicting the expected outcome of future projects undertaken by the organisation (Paulk et al., 1993, p. 3).

2.3.1.3 Software Process Performance

"Software process performance represents the actual results achieved by following a software process" (Paulk et al., 1993, p. 4). Therefore, software process performance focuses on the results achieved and software process capability focuses on the results expected. The actual performance of a project may not reflect the full process capability of the organisation because the capability of the project is constrained by its environment. For example, changes in technology may increase the learning curve of the project's staff. This may prevent the organisation from fully utilising its processing capability.

2.3.1.4 Software Process Maturity

Software process maturity implies a process which has been "... explicitly defined, managed, measured, controlled and effective" (Paulk et al., 1993, p. 4). Maturity suggests a growth in capability. This implies that the organisation's software process has improved and is consistently being practiced in all projects engaged by the organisation. Software process is generally well-understood through documentation and training. The process is constantly being observed and refined by its users. Consistent application of the software process will eventually help improve productivity and quality.

2.3.2 Overview of the Capability Maturity Model

Very often, software engineers and managers are fully aware of their problems but they may not agree on which improvements are most crucial. Without an organised strategy for improvement, it is extremely hard to have an idea on which improvement activities to achieve first. Paulk et al. (1993, p. 5) suggests designing an evolutionary path that will improve an organisation's software process maturity in stages. The software process maturity framework structured these stages so that improvements at each stage will serve as the foundation for improvements for the next. This framework acted as a road-map for consistent process improvement. It does not serve as a "quick-fix" for projects in trouble but rather as a guide for early detection and identifying of deficiencies in the organisation.

The **Capability Maturity Model** (CMM) provides software organisations with guidelines on how to achieve control over their development and maintenance process and how to improve toward accomplishing software engineering and management excellence. The CMM was designed to direct software organisations in selecting the right process maturity and identifying some of the most critical issues that are related to software quality and process improvement (Paulk et al., 1993, p. 5). An organisation can continue to improve its software process by concentrating on this finite set of activities and working assertively to accomplishing them.

The CMM is divided into five maturity levels. Each of these levels define an ordinal scale for determining the maturity of an organisation's software process and for assessing its software process capability. The levels also assist the organisation to prioritise its improvement efforts. Each maturity level accommodates a layer that serves as the foundation for continuous process improvement. Each level also includes a set of process goals when achieved will improve the process capability of the organisation. The five maturity levels are characterised as (Paulk et al., 1993, p. 7):

- *Level 1 - Initial Level*
  At this level, the organisation usually does not have a stable environment for developing and maintaining software. The software process capability at this level is often unpredictable because the software process is often changed as the work progresses. Schedules, budgets, functionality and quality are usually unpredictable too. Performance depends on the capabilities of individuals whose skills, knowledge and motivations varies. Performance can only be determined on an individual basis (Paulk et al., 1993, p. 9).

- *Level 2 - Repeatable Level*
  At this level, procedures for managing a software project, and methods for implementing these procedures are instituted. Experience for planning and managing of new projects is acquired from similar projects. Its objective is to establish an effective management processes for software projects. This will permit the organisations to apply the successful practices that was developed on earlier projects. An

effective process is one that has been practiced, documented, enforced, trained, measured and able to improve (Paulk et al., 1993, p. 10).

Projects at this level are said to have basic software management control. Realistic project commitments are derived from the results gathered from previous projects and from requirements of the present project. The roles of software managers are to track software costs, schedules, and functionality. Software requirements and work products developed to satisfied these requirements are baselined, and their integrity controlled. Software project standards are also defined and the organisation ensures that they are strictly followed (Paulk et al., 1993, p. 10).

- ❑ *Level 3 - Defined Level*

  At this level, the standard process for developing and maintaining software is documented. This includes both software engineering and management processes. These processes are then combined to form a cohesive whole. Processes established at this level are used (and changed, if required) to assist the software managers and technical staff to perform more efficiently.

  Projects tailor the organisation's standard software process to create their own defined software process. This will explain the unique characteristics of each project. This tailored software process will include a cohesive, integrated set of well-defined software engineering and management processes. A well-defined process is one that includes "... readiness criteria, inputs, standards and procedures for performing the work, verification mechanisms, outputs, and completion criteria" (Paulk et al., 1993, p. 11). Since the software process are well-defined, it provides management with an awareness of the technical progress on all projects.

- ❑ *Level 4 - Managed Level*

  At this level, the quantitative quality goals for both software products and processes are established. Productivity and quality are measured to determine any important software process activities. The data gathered are stored and analysed in an software process database. Software processes are equipped with well-defined and consistent measurements. These measurements form the quantitative foundation for assessing the projects' software processes and products. Controls over the products and processes are accomplished by reducing the variation in their process performance so that it falls within the favourable quantitative boundaries (Paulk et al., 1993, p. 12).

- ❑ *Level 5 - Optimising Level*

  At this level, the organisation concentrated mainly on improving its software process. The organisation has the ability to recognise weaknesses and reinforce the process pro-actively. Data on the usefulness of the software process is utilised to carry out cost benefit analyses on new technologies and proposed modification to the organisation's software process. Effective software engineering practices are identified and deployed throughout the organisation. Defects found are analyse to determine their causes. Software processes are assessed to prevent known defects from repeating and the lesson learned are administered onto future projects. The main objective of the organisation is to continue improving their process capability, in effect, improve the process performance of their projects (Paulk et al., 1993, p. 13).

The CMM is a model that describes the main attributes that would be expected to characterise an organisation at a particular maturity level. The CMM is described at an adequate level of abstraction so that it does not unnecessarily constrain how the software process is implemented. The CMM must be properly interpreted, based on informed professional conclusion. Paulk et al. (1993, p. 14) pointed out that the CMM does not explicitly instruct an organisation on how to improve. It merely describes an organisation at each maturity level. He also added that it usually takes a couple of years (maybe more) for an organisation to move from one level to the next.

2.3.3 Future Directions Of The CMM

The CMM is not the solution to all problems. It does not cover all the issues that are vital to the success of a project. According to Paulk et al. (1993, p. 51), CMM presently does not address "expertise in particular application domains, advocate specific software technologies, or suggest how to select, hire, motivate, and retain competent people". Although these issues are important to the success of a project, some of them have been analysed in other contexts. Unfortunately, they have not yet been incorporated into CMM. The CMM was intentionally developed to provide an systematic, disciplined framework so that it can address software management and engineering process issues.

2.4 FUNCTION POINT ANALYSIS

Allan Albrecht was looking for a method of measuring productivity in software development. Realising that the *line of code* approach was not very reliable, Albrecht wanted to develop an alternative method. Hence in 1979, he developed the function points model (Heemstra et al., 1991, p. 230). As the name suggests, this model counts function points, as opposed to the very popular lines of code model. In fact, function point analysis is conducted even before coding begins. Function points relate directly to the client's requirement in a way that is more easily understood by the client than SLOC (Albrecht et al., 1983, p. 639). Function points can also be used as a general measure of development productivity, which may be used to illustrate productivity trend (See **Section 2.2.3.2**).

In addition, function point analysis does not count functions that are found to be necessary by the programmer but were not specifically requested by the user. Therefore, a function point is regarded as one end-user requested function (Grupe et al., 1991, p. 24). For example, if a user requests that this month's sales figures be retrieved from a data base, that request becomes one function point.

After it was first developed, the function points model was later revised by Symons into what was later known as the ***Mark II*** (see **Section 2.4.4**) function points model (O'Brien et al., 1993, p. 3). Although many consider function point analysis to be a relatively new concept, it has arose as an important methodology for estimating and validating the limits and size of a software project. With this knowledge, it is possible to measure productivity and the influence of various tools and procedures (Kizior, 1993, p. 42). Kizior (1993, p. 42) added that function point analysis is not used to measure work input, quality, or value to the user.

Though the importance of function point analysis has been recognised, it has not been well publicised. This is found to be the case when Kizior (1993, p. 42) conducted a review on textbooks published within the past eight years which deal with software design, systems analysis and design, and general information on system concepts. Kizior (1993, p. 42) found that of the 32 books reviewed, only two made explicit mention of function point analysis.

2.4.1 Advantages And Disadvantages Of Function Point Analysis

Function point analysis has become popular within the last several years due to its inherent advantages. These advantages are (Kizior, 1993, p. 45):

- ❑ Function point analysis measures function that is delivered to the user
- ❑ It is not dependent on hardware and software
- ❑ It is reliable early in the design cycle to aid the estimating process
- ❑ It can be meaningful to the end user

Having listed the advantages, the accuracy of counting function points is proportional to the knowledge of the person counting. According to Kizior (1993, p. 46), counting function points cannot be considered as a science because some subjective judgements had to be made. Furthermore, Ratcliffe and Rollo (cited in O'Brien et al., 1993, p. 3) showed that the count achieved is dependent on the notation used to describe the software requirements. In addition, it was found that experienced analysts were more accurate in function point count than those without a notable level of experience (Graham et al., 1990, p. 71). It is also fair to say that it does not make anyone proficient in counting function points simply by undertaking a function point training course. Beginners should be assisted for a period by an experienced analyst so that they may be able to achieve consistent results. Other drawbacks of function points are that they cannot (Kizior, 1993, p. 46) :

- ❑ Measure individual effort
- ❑ Measure productivity (only to a certain degree)
- ❑ Measure quality
- ❑ Measure value to user

Ferens et al. (1992, p. 641) state that the function points method is not readily suited for real-time or scientific environments. They did, however, briefly mention that authorities such Capers Jones, Donald Reifer, and John

Gaffney and Richard Werling are attempting to adapt the function points concept into these environments. They also added that little independent research has been done on real-time variations of function points. Therefore, it is difficult to ascertain whether function points can be useful outside the data processing environment.

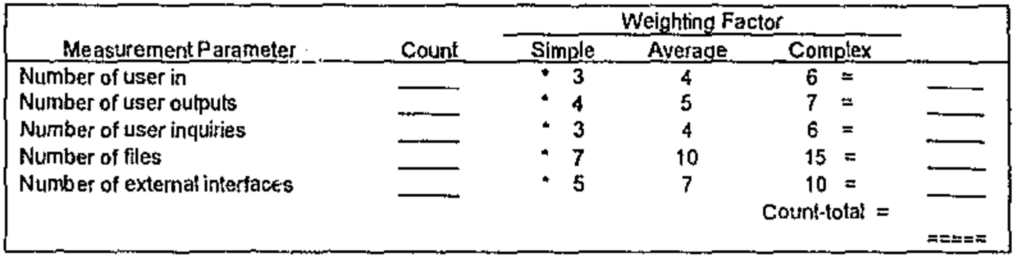
2.4.2 Counting Function Points

The principle of function point analysis is simple. It is based on the number of functions that are delivered in the final system. The general assumption is that the more function points an application has, the more complex system becomes (Grupe et al., 1991, p. 24). The more complex the system, the longer it will take and the more expensive it becomes to develop the system.

Simply put, function point analysis is a weighted sum of five primary end-user function-related attributes. The function points that are identified during system analysis are grouped into five categories which will be adjusted by a complexity factor (Grupe et al., 1991, p. 24). These categories are :

- ❑ the external input type (for example : mouse input)
- ❑ the external output type (for example : viewing items on a screen)
- ❑ the external inquiry type (for example : accessing a record without update)
- ❑ the logical internal file type (for example : master and transaction files)
- ❑ the external interface file type (for example : sharing files with other applications and external files)

Albrecht et al. (1983, p. 639) pointed out that "... these factors are the outward manifestations of any application. They cover all the functions in an application. Each of these categories of function types are counted individually and then weighted by numbers reflecting the relative value of the functions to the user/customer". ***Function points*** is the weighted sum of these function types. Organisations that use function point methods often develop criteria for determining whether a particular entry has a simple, average or complex weighting factor (See **Figure 2.4.2.1**). According to Albrecht et al. (1983, p. 639), the weighting factors used were "determined by debate and trial". And as mentioned before, the determination of the complexity of these function types is somewhat subjective.



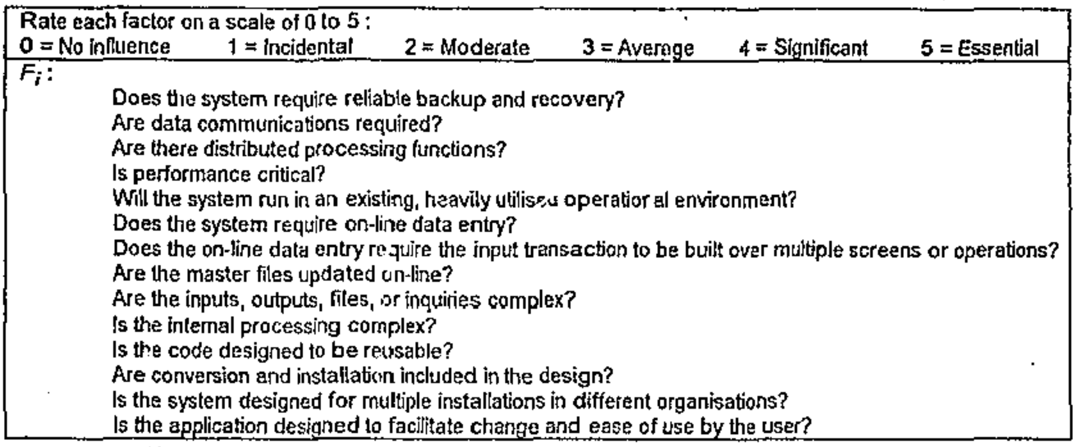
		Weighting Factor			
Measurement Parameter	Count	Simple	Average	Complex	
Number of user in	____	* 3	4	6 =	____
Number of user outputs	____	* 4	5	7 =	____
Number of user inquiries	____	* 3	4	6 =	____
Number of files	____	* 7	10	15 =	____
Number of external interfaces	____	* 5	7	10 =	____
				Count-total =	=====

Figure 2.4.2.1 **Table - Computing Function Point Metrics**

To calculate function points, the following equation is used (Pressman, 1992, p. 49) :

$$FP = \text{Count-total} * [0.65 + (0.01 * \text{SUM}\ (F_i))]$$

where Count-total is the sum of all FP entries obtained from the table in Figure 2.4.2.1. F_i (where i = 1 to 14) are *complexity adjustment values* based on the responses to questions listed in **Figure 2.4.2.2**. The constant values in the above equation and the weighting factors that are applied to information domain counts are determined empirically.



Rate each factor on a scale of 0 to 5 :

0 = No influence	1 = Incidental	2 = Moderate	3 = Average	4 = Significant	5 = Essential

F_i :

Does the system require reliable backup and recovery?
Are data communications required?
Are there distributed processing functions?
Is performance critical?
Will the system run in an existing, heavily utilised operational environment?
Does the system require on-line data entry?
Does the on-line data entry require the input transaction to be built over multiple screens or operations?
Are the master files updated on-line?
Are the inputs, outputs, files, or inquiries complex?
Is the internal processing complex?
Is the code designed to be reusable?
Are conversion and installation included in the design?
Is the system designed for multiple installations in different organisations?
Is the application designed to facilitate change and ease of use by the user?

Figure 2.4.2.2 **Table - Computing Function Points - Complexity Adjustments Values**

Once the function points have been calculated, they can be used as a measure of software productivity, quality, and other attributes. For example :

Productivity	= FP / person-month
Quality	= defects / FP
Cost	= $ / FP
Documentation	= pages of documentation / FP

2.4.3 Function Point Analysis : An Evaluation

The function point metric, like the *lines of code* metric, is controversial. Those that are *for* the function point metric, claim that function points are programming-language independent. Hence, making it suitable for applications using conventional and non-procedural languages. Proponents also claim that function points is more attractive as an estimation tool because estimation can be made early in the life-cycle of a project (Pressman, 1992, p. 51).

On the other hand, the opponents are claiming that the function point metric requires some "sleight of hand" because some part of the computation is based on subjective rather than objective data (Pressman, 1992, p. 51). That is to say, when two individuals are performing a function point count on the same system, they may not come up with the same number of function points.

They also claim that function points have "... no direct physical meaning ..." because they are only numbers (Pressman, 1992, p. 51).

To determine whether function point analysis is indeed as good as it is claimed to be, Heemstra and colleague (1991, p. 229) performed a series of studies. The studies include an analysis based on the data from a large survey of Dutch organisations, from an experiment regarding the use of software cost estimation models and from a field study aimed at the adjustment factor of the function point analysis model. The questions that Heemstra et al. (1991, p. 229) were attempting to answer are :

- □ Is function point analysis actually used in practice?
- □ How is function point analysis used in practice?
- □ How reliable are the estimates made with function point analysis?
- □ Are models based on function points better then models based on lines of code?
- □ How effective are the function point analysis adjustment characteristics?

The report produced by Heemstra et al. (1991, p. 236) (based on their data from the survey of Dutch organisations) confirmed that function point analysis is indeed widely used in the Netherlands. If this model became a standard tool, it could provide organisations with necessary information of their previous experiences so that they could learn from them in a methodical way. However, Heemstra's report also showed that using this tool alone will not resolve all the problems in this area.

From the experiment, function point analysis performed quite well as a tool for measuring size. Its result superseded the lines of code method as an estimator within the setting of Heemstra's experiment. This also proved to be the case in a study conducted by Graham et al. (1990, p. 71). In Graham's study, function point analysis also proved to be more consistent than the line of code method. Hence, the results confirm that function point analysis is a more acceptable metric for measuring software size.

However, the results from the field study regarding the adjustment part of the model is less than satisfactory. Experienced users showed no confidence at all in the adjustment characteristics. In Heemstra's (1991, p. 236) experiment, there were many disagreements against the notion of a small set of generally applicable cost drivers. Heemstra concluded that precaution measures must be considered when using any model. After all, a model is not a machine where questions are fed from one end and the correct answers produced from the other end.

Function points have proved to be a broadly popular measure with both practitioners and academic researchers. According to Dreger (cited in Kemerer, 1993, p. 87), it is estimated that there are around 500 major corporations worldwide currently using the function point analysis method. Graham et al. (1990, p. 65) also state that function points are currently being used by numerous large Australian organisations to measure productivity for project review purposes and effort estimation. And according to a survey conducted by the Quality Assurance Institute (Kemerer, 1993, p. 87), the function point method was found to be the best available MIS productivity measure. In addition, Ferens et al. (1992, p. 641) pointed out that the International Function Points User's Group (IFPUG) has been formed to continually improve the function points theory and practice. The IFPUG is also studying and revising some of Albrecht's equations.

2.4.4 Mark II Function Point

The aim of the Mark II approach was to overcome some of the weaknesses of Albrecht's function point approach. However, Symons (1988, p. 8) pointed out that there will never be any evidence that the Mark II approach will give higher results to that of Albrecht. With the Mark II approach, the methods for counting data elements has been introduced to make the complexity classification of inputs, outputs and entities more objective. The concept of "logical files" has replaced by "entities". This means that instead of having five attributes like Albrecht's method, Mark II only has three : inputs, outputs and entities (Ferens et al., 1992, p. 641). The Mark II approach assigns Unadjusted Function Point's (UFP) to data based on its usage (create, delete, etc.) in transactions, whereas Albrecht's approach will assign UFP's to all the data that exist in the system (Symons, 1988, p. 8).

Symons (1988, p. 8) pointed out some of the differences or similarities between Mark II and Albrecht's function point model as :

- ❑ The Mark II approach requires an understanding of entity analysis and the rules for entity counting is now available. In Albrecht's approach, knowledge of entity analysis is desirable but it has no entity counting conventions yet.
- ❑ The Mark II approach has fewer variables in the UFP component. Hence, it has a number of advantages such as greater ease of calibration against measurements and estimates.
- ❑ Even though this theory has not yet to be examined, according to Symons, the Mark II approach has the capability of improving the measurement of the work-output in the maintenance and enhancement activities. Albrecht's approach can only measure the total size of a changed component, without distinguishing on how big or small these changes are. The Mark II approach can measure the size of the changes

made to a component, if the number of data elements changed are recorded and the references to these changed entities are accounted for.

- The Mark II approach may require about 10 to 20 percent more effort (than Albrecht's approach) for counting each input and output data elements. This suggests that Albrecht's approach may be applied slightly earlier in the project life-cycle. Symons (1988, p. 9) believes that it may still be able "... to produce reasonably accurate estimates of the number of data elements per transaction for early sizing purposes".

2.5 UNDERGRADUATE SOFTWARE ENGINEERING PROGRAMS

According to Grant et al. (1991, p. 106), the state of Software Engineering practice in Australia is still generally rather primitive. It is believed that educational institutions such as Edith Cowan University have a major role to play in the transformation of this practice. What is needed by these educational institutions are degree programs with a strong emphasis on Software Engineering. The computing curricula in Australia tend to have an emphasis either in Computer Science, Information Systems or Computer Systems Engineering. There is a need to develop a curriculum with a strong Software Engineering emphasis. There must be a fair balance of both theoretical and practical technical foundations. Furthermore, it is believed that with careful planning and direction, software engineering projects can provide students with an opportunity to experience how software is being developed in the real-world (Shaw et al., 1991, p. 33). It is very difficult to define a completely satisfactory curriculum, because software engineering has yet to reach the stage of being a mature engineering discipline.

2.5.1 Objectives Of Software Engineering Courses

The previous section pointed out the need for software engineering programs in educational institutions. This section will discuss the main objectives of these programs, from the students' point of view. When undertaking such programs, the students are expected to (Grant et al., 1991, p. 107) :

- ❑ develop adequate technical skill in analysis, design and programming
- ❑ understand the primary concepts of Software Engineering
- ❑ develop and/or improve their inter/intra personal skills so that they can participate in a software development team
- ❑ participate in practical work that requires the understanding and use of these concepts and skills
- ❑ appreciate (through experience) the benefits of methodological approaches to systems development and the consequences of ad hoc approaches

Students must understand and accept the benefits of undertaking a practical software engineering project. Therefore, educational institutions should provide students with a learning environment where students can experience and learn the important role that methodology plays in the success of the project. Hence, the final year software engineering project is technically complex that requires a high degree of communication and control. It is believed that the project will definitely fail (or not up to standard) if it is "... approached in an ad hoc manner" (Grant et al., 1991, p. 108). However, it is not easy to select a one-year software engineering project. As mentioned before, the project is technically complex but at the same time, it should not be too complex that it cannot be completed in two semesters. It must be made clear to the students that such a project is to be treated as a software engineering project and not a programming assessment (Adams, 1993, p. 112).

A well-defined project will provide adequate time to demonstrate to students the need for software engineering disciplines and the approach to managing complex projects. The students must put into practice the theories they have acquired, such as group organisation and project management. (Grubb, 1991, p. 2). .

The projects that are provided by the Department of Computer Science of Edith Cowan University require a lot of team work and communication among students, staff advisers and the *client(s)*. The students are required to work in teams of 4 to 5 members each. Group projects play an important role in many software engineering courses. As Calliss et al. (1991, p. 25) suggest, "factors, such as group dynamics, egoless programming and team organisation, that affect the way programmers work together cannot be taught effectively in a classroom settings". The students must experience the problems of working in a group (Briggs, 1991, p. 48) because this will serve as an important step towards the students' appreciation of the solutions to these problems.

The group project was designed so that it required students to communicate with each other, their staff adviser and the client. The most common form of communication is through group meetings. Although there is no penalty for students who are absent from group meetings, it is expected that they establish some form of group communication either written, verbally or electronically. It is an objective of the Edith Cowan University Computer Science department that students can learn the benefits of effective communication and the consequences of poor communication. According to Grant et al. (1991, p. 108), there is sufficient "... anecdotal evidence that concentration on communication skills has provided the behavioural and social transformations in computing graduates most appreciated by employers in recent years."

The development methodology enforced by the university is the APT (EXECOM, 1991) methodology. The APT methodology is based on the waterfall model. Though students working on the software engineering project are free to select other types of methodologies, the majority of students still use APT. However, data[10] gathered from the 1993 software engineering students showed that the APT methodology was not very suitable in many cases. Nonetheless, this model serves as a good learning methodology from the students' point of view.

[10] From a study conducted as part of this thesis.

3.1 OVERVIEW OF THE ORCHARD PROJECT

The 1993 software engineering project was called the Orchard Project. The client for this project was Mrs Vivian Campbell, who is a lecturer of Edith Cowan University, Bunbury campus and is also an orchardist. The aim of this project was to develop a software system which would allow orchardists and horticulturalists to formulate an efficient farm management strategy ("Orchard", 1993).

The students undertaking this project were required to use the tools and techniques acquired in their course to analyse requirements and data. This will enable the students to produce a system that provides orchardists with a means to identify and collate all the vital areas of orchard operations. These operations include the identification of optimal fruit varieties, staff management, farm infrastructures and create efficient marketing strategies ("Orchard", 1993).

The students were also encouraged to develop the database so that it would meet the orchardist's other requirements. These requirements included keeping detailed insecticide spray and fertiliser records, irrigation schedules and identifying which fruits are most profitable on the local and international markets ("Orchard", 1993).

3.2 GOALS OF THE ORCHARD PROJECT

According to the client, Mrs Campbell, the orchard management system should be able to provide the orchardist with essential information such as tax and superannuation, and should also provide information that will aid the orchardist in making management decisions such as purchasing and hire of workers. The goals of the system was to aid the orchardist in making a greater profit and producing excellent fruit for the local and overseas markets.

3.3 MAIN ASPECTS OF THE MANUAL SYSTEM

There are six main aspects in Mrs Campbell's orchard business. These aspects are :

- □ Fruit production
- □ Taxation
- □ Other farm
- □ Marketing
- □ Staff management
- □ Research

3.3.1 Fruit Production

Fruit production deals mainly with the growing and maintaining of trees. The activities that are associated with it, are :

- ordering of new trees
- planting of trees
- pruning and training trees
- application of fertilisers and sprays
- fruit thinning
- fruit picking and packing
- irrigation

Other aspects that are also involved in fruit production includes fencing, pest control, mowing and weed control and machinery maintenance.

3.3.2 Marketing

Marketing includes recording of sales information for both local and overseas markets. The sales information records the quantity of the various fruits sold as well as its price. However, in the local situation, the prices of these fruits vary from day to day. Accordingly the orchardist has to be well aware of the current prices. The orchardist will also need to maintain information regarding the crates and disposable trays used, for they all have monetary values.

3.3.3 Taxation

In taxation, the primary concern is keeping the business' accounts up to date. The accounts are divided into income and expenditure. Those that are classified as income are fruits sold, other farm income (see **Section 3.3.5**) and bank interest. The fruits sold are categorised by variety (eg. apples and peaches etc). The other farm categories are income derived from the sale of wool and livestock.

The expenditure accounts are categorised as follows :

- labour
- pesticides
- trees
- electricity
- bank charges
- cartage
- fruit packaging
- fertiliser
- herbicides
- insurance
- rates
- machinery repairs
- hire of machinery

3.3.4 Staff Management

Staff management includes the hiring and firing of employees, calculating and paying of employees' wages, and calculating and paying of employee's superannuation. The wage of an employee is calculated based on the employee's job type, age, mode of employment and hours worked.

3.3.5 Other Farm

Other farm aspects include stock control on items such as fertiliser, pesticides etc, materials used for fencing and water storage. It also includes livestock management, mainly related to sheep.

3.3.6 Research

Research mainly involves the identification of new varieties of trees and fruits, and new methods for maintaining the growth of the trees and fruits.

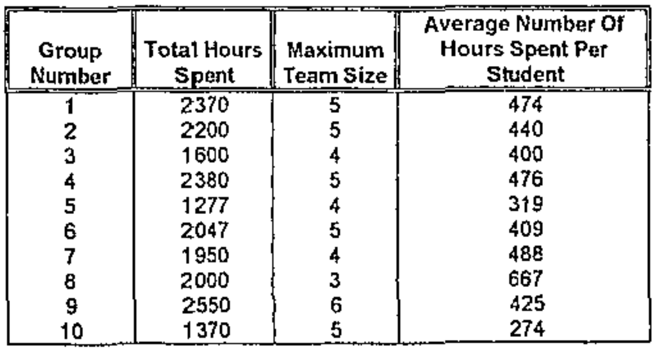
3.4 REQUIREMENTS OF THE NEW SYSTEM

The new system should be able to perform all of the crucial tasks mentioned in the previous section. The client has identified those tasks as being :

- ❑ calculate the taxes based on information stored in the income and expenditure accounts.
- ❑ identify the variety of trees that are the least or most profitable
- ❑ create a budget and to project cash flow
- ❑ identify sales trends based on year to year comparison of costs
- ❑ maintain records on which sprays and fertiliser are being applied
- ❑ irrigation scheduling

4.1 TOTAL HOURS SPENT ON THE PROJECT

This section presents the total number of hours spent on the project by each group. The data were initially gathered during a pilot study, which lasted for 14 weeks. The data were collected on a weekly basis in the form of questionnaires. Students were asked to log the number of hours spent on the project for the week. Since it was not mandatory for the students to take part in this research project, a major portion of the data gathered were inconsistent and incomplete. Therefore, a second set of questionnaires were prepared. These questionnaires were given to the students after their project demonstration. This was to ensure that all the students for each group were accounted for. It was mandatory that all students participate in this exercise. Students were requested to answer the questionnaires to the best of their ability and they were requested not to confer with each other. The data gathered are presented in **Figure 4.1.1** and **Figure 4.1.2**.



Group Number	Total Hours Spent	Maximum Team Size	Average Number Of Hours Spent Per Student
1	2370	5	474
2	2200	5	440
3	1600	4	400
4	2380	5	476
5	1277	4	319
6	2047	5	409
7	1950	4	488
8	2000	3	667
9	2550	6	425
10	1370	5	274

Figure 4.1.1 **Table - Total Number Of Hours Spent On The Project**

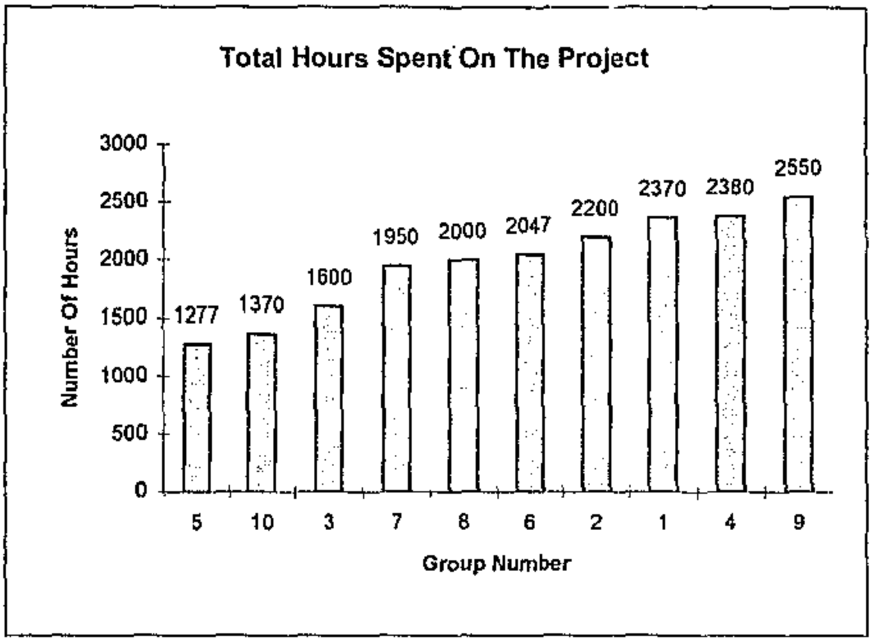


Figure 4.1.2 **Graph - Total Hours Spent On Project By Each Group**

4.2 DATA COLLECTED FROM THE RESEARCH QUESTIONNAIRE

From the second set of questionnaires (as mentioned in the previous section), other types of data were also collected. These were :

- ❑ The number of hours spent on each phase of the development life-cycle. In this case, the life-cycle included requirement, analysis, design, coding and testing phases.
- ❑ The personal attributes of each member of a group - age, gender and study mode
- ❑ The quality of project management.
- ❑ The usefulness and effectiveness of the APT (EXECOM, 1991) methodology.
- ❑ The effectiveness of having a staff adviser.
- ❑ The effectiveness and usefulness of the product(s) used to develop the software.
- ❑ The quality of user requirements obtained from the client.
- ❑ The effectiveness of working as a team.
- ❑ The quality of contribution made by each team member.
- ❑ The ability to meeting deadlines.

4.2.1 Effort On Each Development Phase

The effort employed in each phase of the development life-cycle was broken down into five phases - requirement, analysis, design, coding and testing. The details of effort collected from each student was expressed in percentage terms. The data for each group were then totalled and averaged to determine the effort (in percentage) for each phase. The results are presented in **Figure 4.2.1.1**.

Group	Breakdown Of Effort Per Group (%)					
	Requirement	Analysis	Design	Coding	Testing	Total
1	16	20	23	24	18	100
2	23	26	19	18	14	100
3	9	13	10	54	15	100
4	14	26	25	17	19	100
5	10	21	18	38	14	100
6	16	19	15	36	14	100
7	17	24	22	25	13	100
8	18	23	30	19	10	100
9	12	24	23	14	27	100
10	18	31	16	24	11	100
Average :	15	23	20	27	15	100

Figure 4.2.1.1 **Table - Effort On Each Phase (In Percentage)**

The same information in **Figure 4.2.1.1** is translated into number of hours spent on each phase. This calculation is derived using the total number of hours obtained from **Figure 4.1.1 - Total Number Of Hours Spent On The Project**. The results are presented in **Figure 4.2.1.2** and **Figure 4.2.1.3**.

Group Number	Breakdown Of Effort Per Group (In Hours)				
	Requirement	Analysis	Design	Coding	Testing
1	387	465	543	558	418
2	506	572	418	387	317
3	140	200	160	860	240
4	338	609	585	407	440
5	128	271	223	479	176
6	325	389	307	739	287
7	332	463	424	488	244
8	353	467	600	380	200
9	305	603	575	367	700
10	249	425	216	332	148
Average	306	446	405	500	317
Minimum	128	200	160	332	148
Maximum	506	609	600	860	700

Figure 4.2.1.2 **Table - Total Hours Spent On Each Phase By Each Group**

Group	Breakdown Of Effort In Average Per Student-Group (In Hours)				
Number	Requirement	Analysis	Design	Coding	Testing
1	77	93	109	112	84
2	101	114	84	77	63
3	35	50	40	215	60
4	68	122	117	81	88
5	32	68	56	120	44
6	65	78	61	148	57
7	83	116	106	122	61
8	118	156	200	127	67
9	51	100	96	61	117
10	50	85	43	66	30
Average	68	98	91	113	67
Minimum	32	50	40	61	30
Maximum	118	156	200	215	117

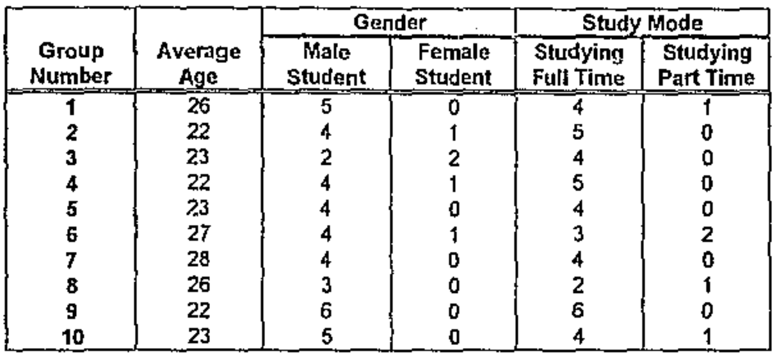
Figure 4.2.1.3 **Table - Average Hours Spent On Each Phase Per Student-Group**

Judging from the tables above, the majority of the groups spent more time on Analysis and Coding and less time on Requirement and Testing. It was found that most of the groups went out into the industry to conduct their own research on methods for calculating tax, and gaining more information on the operations of an orchard business. This is reflected in the amount of time spent on Analysis. As for Coding, the software used to developed the application were relatively new (except for Objectvision Pro). This lack of previous exposure to the software obviously contributed to an increase in the time required to complete coding.

As for Requirement, students spent the least amount of time on this. This was probably due to the fact that the client could not be reached by the students directly and the user requirements were provided in two information-gathering sessions (each lasted for about one hour). However, some teams did further their research among local orchardists. Students also spent less time on Testing. The assumption being that since the majority of the projects were behind schedule, their software was not fully tested. However, during the demonstration of these projects, most of the groups told the judging panel that extensive testing was indeed conducted. Based on the information presented in **Section 6.2.3 - Evaluation Report**, it shows otherwise. It would appear that the software testing was limited to the various modules rather than the whole systems.

4.2.2 Personal Attributes Of Group Members

Figure 4.2.2.1 presents the composition of the groups based on the students' age, gender and study mode.



Group Number	Average Age	Gender		Study Mode	
		Male Student	Female Student	Studying Full Time	Studying Part Time
1	26	5	0	4	1
2	22	4	1	5	0
3	23	2	2	4	0
4	22	4	1	5	0
5	23	4	0	4	0
6	27	4	1	3	2
7	28	4	0	4	0
8	26	3	0	2	1
9	22	6	0	6	0
10	23	5	0	4	1

Figure 4.2.2.1 **Table - Personal Attributes Of Each Group**

The average age of all the students was around 24 years old. Out of the 41 students, 12 per cent were female students and 12 per cent were part-time students.

4.2.3 Staff Adviser

Each group was assigned a staff adviser, whose role was to act as a consultant to the group members. **Figure 4.2.3.1** and **Figure 4.2.3.2** presents the rating (out of 10) that students gave for their staff adviser, and the total number of times the students met with their staff adviser. The "meetings" that these students had could be group or individual meetings. Student meetings with staff advisers were not compulsory under the project guidelines.

Group Number	Staff Adviser's Rating	Meeting With Staff Adviser
1	4.2	15
2	6.6	2
3	2.8	0
4	8.2	10
5	5.0	7
6	6.0	8
7	2.0	6
8	3.3	4
9	7.0	25
10	6.2	20

Figure 4.2.3.1 **Table - Staff Adviser**

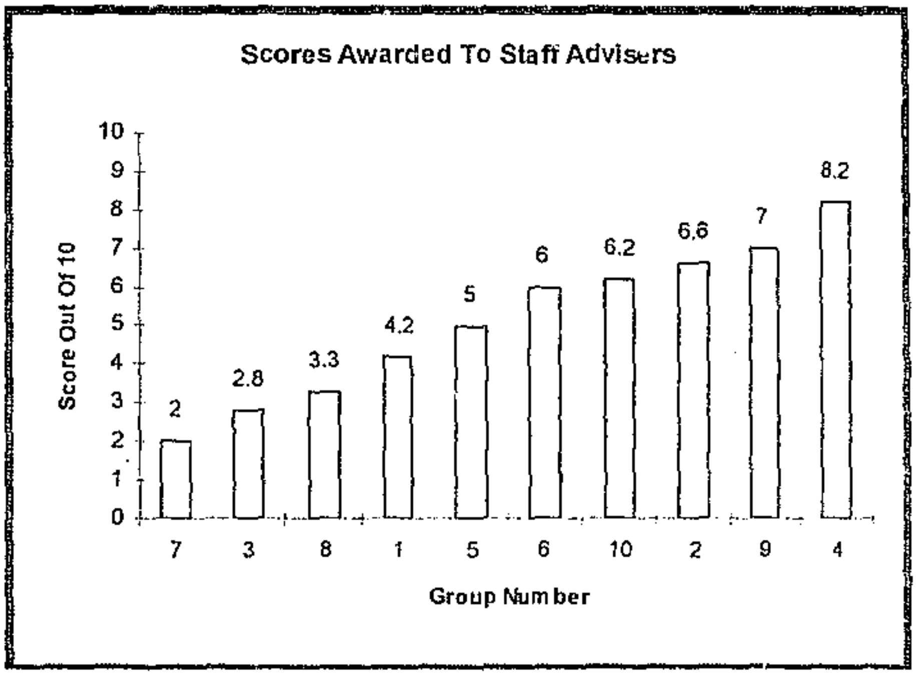


Figure 4.2.3.2 **Graph - Scores Awarded To Staff Adviser By Students**

The results in **Figure 4.2.3.1**, indicate that some groups have minimal or no interaction with their staff adviser.

It is also important to point out that **Group 5** and 7 had two different staff advisers. Their first staff adviser left some time during the middle of the first semester. These groups were then reassigned to another staff adviser (**Staff Adviser 3**). It is not clear the number of meetings these groups had with the respective staff advisers. For example, **Group 5** stated that they had about 7 meetings with their staff adviser but their staff adviser (**Staff Adviser 3**) did not mention any meeting he had had with the group. Of course, all these data were based on the individual recollections of the event and **Staff Adviser 3** was responsible for 4 groups.

In conforming with the university's Ethical policies, the name of each staff adviser will remain anonymous. Therefore, each staff adviser was assigned a unique number. **Figure 4.2.3.3** shows which group(s) were assigned to which staff adviser. The data represented in *italics* were those groups whose software was not evaluated because they could not be made operational. Despite the fact that all software was operational for the assessment presentation, it was only possible to get 10 of the 16 working for subsequent analysis. However, the data from staff advisers for these groups was taken into consideration.

Staff Adviser	Group Number
1	1
2	2
3	3
3	5
3	7
3	10
4	4
5	6
6	8
7	9
8	*11*
9	*12*
10	*13*
11	*14*
12	*15*
13	*16*

Figure 4.2.3.3 **Table - Staff Adviser For Each Project Group**

To better understand the relationship between staff advisers and students, as well as their opinion of being appointed staff adviser, the staff advisers were interviewed on a structured basis. The results from these interviews are as follows:

- **Staff Adviser 1 :**
  This staff adviser had general knowledge about the APT methodology. According to him, he had meetings with his group once every two weeks during the first semester. Each meeting lasted for about half an hour. During semester two, he met with his group three times, each lasted for about half an hour. He commented that it was a good idea to have a staff adviser assigned to each project group. He claimed that this would provide students with a "contact point", so that students could come for help if they were having problems (that were related to the project).

- **Staff Adviser 2 :**
  This staff adviser knew very little about the APT methodology. He said that he saw his group about three times during the first semester and not al all during the second semester. He commented that having a staff adviser for each group was essential because students "need to have access to a staff member".

- **Staff Adviser 3 :**
  This staff adviser was not very familiar with the APT methodology. He said that there was no fixed time or date for meetings with his groups. According to him, he did meet with Group 10 for about half an hour per week (for 14 weeks), 2 meetings with Group 3 for about half an hour each, during semester one. During semester two, he had 4 meetings with Group 15 for about an hour each. It is important to note that the staff adviser for Group 15 was Staff Adviser 12. He commented that by assigning members of the staff of the department as staff advisers did not really emulate a real-world software development environment. This is a cause for concern, as the aim of the software engineering project was to provide students with "real-world" experience.

❑ **Staff Adviser 4** :

This staff adviser was not very familiar with the APT methodology. He met with the project leader once every two to three weeks during semester one and two. He did meet with the whole group once, towards the end of semester one. He commented that members of the staff should volunteer to become a staff adviser. Since staff members were being assigned to be staff adviser, the department should at least provide some form of training so that the staff adviser would know what to do and what to expect. That way, the staff adviser will be more beneficial to the group.

❑ **Staff Adviser 5** :

This staff adviser was familiar with the APT methodology but had no in-depth knowledge. He said that his group never set up any meetings with him. All he received from the students were progress reports (once every 2 to 3 months). He further added that students should take the initiative of setting up meetings and not the other way around. He said that looking at the progress report was not adequate. He claimed that personal contact was important if a staff adviser was to evaluate the group's progress. He commented that it was important to have someone supervise the student but it would be more effective if members of the staff were willing and interested, instead of just assigning them to groups.

❑ **Staff Adviser 6** :

This staff adviser was not very familiar with the APT methodology. The meetings between students and the staff adviser were very rare. He claimed that the students worked independently. He did not offer any advice or opinions on matters related to the project. He had no background in software engineering and was not fully aware of the project's requirements.

❑ **Staff Adviser 7** :
This staff adviser had about 6 meetings with the group, each lasted between 30 minutes to an hour. The staff adviser also provided the students with presentation skills (2 - 3 hours), lectures on entity-relationship modelling (4 hours) and advice on designing a better user interface (2 - 3 hours).

❑ **Staff Adviser 8** :
This staff adviser had reasonable knowledge on the APT methodology. He did offer his students advice and opinions at the beginning of the project. He had about 4 meetings with the students, each lasting for about 12 minutes. Due to the lack of meetings with the students, he was not aware of the students' progress. He commented that he came from a different discipline and had no knowledge in System Analysis and Design and because of his lack of background knowledge, he was of no real assistance to the students.

❑ **Staff Adviser 9** :
This staff adviser had an average knowledge on the APT methodology. He met with his group once every two weeks during semester one, but in semester two, he did not have any meetings with his group.

❑ **Staff Adviser 10** :
This staff adviser had a fair knowledge of the APT methodology. He said that during the semester one, he met with the project leader about four times, each meeting lasted from 15 to 45 minutes. During semester two, he again had about four meetings with the project leader, but each lasted only from 2 to 5 minutes.

- ❑ **Staff Adviser 11 :**
  This staff adviser was very familiar with the APT methodology. He had very little contact with his group. When they did meet, the students were often poorly organised or not well prepared. He thought that having a staff adviser for each group was a good idea because it worked out quite well with the previous years' projects.

- ❑ **Staff Adviser 12 :**
  There is no information on how familiar this staff adviser was with the APT methodology. The staff adviser said that it would help the student greatly if the role of the staff advisers was clearly defined.

- ❑ **Staff Adviser 13 :**
  There is no information available from this staff adviser.

4.2.3.1 Summary

Based on the information and comments from the staff advisers, the following can be concluded :

- ❑ The staff advisers should have a reasonable amount of knowledge regarding the standard software development methodology adopted by the Computer Science department. This would ensure that they know what to expect from the students.
- ❑ Meetings with students on a regular basis should be made mandatory so that staff advisers are aware of their problems and progress.
- ❑ Staff advisers should be interested and volunteer for the role. This way, the staff adviser will be more interested in the progress and development of the group project.
- ❑ Staff advisers should have sufficient knowledge of the software development process.

- ❑ Staff advisers should have a clear understanding of the nature of the software engineering project.

If the criteria mentioned above are satisfied, it should ensure that all groups will have a conipetent degree of supervision from their staff adviser. Then students can really have a taste of what the "real-world" situation is like. It is true that in certain cases, students were experiencing "real-world" problems (eg. difficulty in meeting deadlines) with their project. Some would argue that not all human beings are the same, hence not all staff advisers are the same, but from the students' point of view, they were being assessed on their project. The role of the staff adviser should be an added advantage rather than a disadvantage to the students. In addition, supervision should be consistent across all teams.

Staff advisers 4 and 7 scored very well from the students they supervised. Staff adviser 4 maintained a consistent meeting schedule with his group's project leader. By doing so, the amount of interaction between the staff adviser and student was high. Staff Aɹviser 7 seems to have taken a more active role with the students by providing them with more in-depth guidance. From these, it is clear that the high level of student-staff adviser interaction, has earned them the highest ratings. However, it is interesting to note that Group 2 only met with their staff adviser (Staff Adviser 2) about 2 to 3 times and yet, they awarded a score of 6.6 (the third highest) for their staff adviser. This is important to point out that there is an element of students not wanting to say anything negative about senior members of the department, which contributes to the distortion of results.

4.2.4 Development Software Used

This section presents the type of software that each group used to develop their software. The students were also asked to provide a rating (out of 10) for the software they used. When questioned on the reason for their selection, the majority stated that their software had received positive reviews from computer articles and magazines. The data are presented in **Figure 4.2.4.1** and **Figure 4.2.4.2**.

Group Number	Development Software Used	Usefulness Of The Software
1	Microsoft Access	6.0
2	Microsoft Access	5.6
3	Paradox For Windows	5.0
4	Microsoft Access	8.5
5	Microsoft Access	6.8
6	Paradox For Windows	5.0
7	Microsoft Access	5.5
8	Microsoft Access	6.3
9	Microsoft Access	5.5
10	Objectvision Pro	3.8

Figure 4.2.4.1 **Table - Development Software Used**

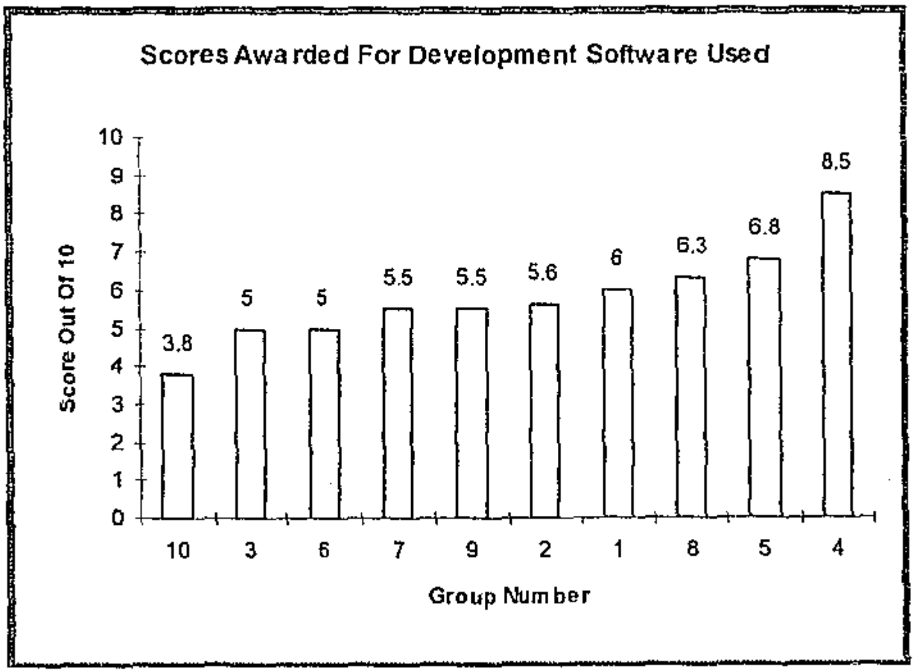


Figure 4.2.4.2 **Graph - Scores Awarded For Development Software Used**

All development software selected were relatively new in the market. Since all these packages operated in the Windows environment, they promised screen design facilities, and importing of colourful graphic images. All the software selected provided some form of 4GL-like tools to help make design easier. The key word here is "design". When Microsoft released Access Version 1.0 and subsequently Version 1.1, their aim was to provide **end-users** with capabilities to design their own applications. The same applies to Paradox for Windows. These packages have integrated form and report design facilities. As for Objectvision Pro, its early counterpart Objectvision was very similar to Microsoft Access. Borland promised that users could develop applications with Objectvision without any programming. With Objectvision Pro, Borland added a report generator and programming language (Turbo C++) to make it more powerful. Each piece of software has its own pros and cons, as reviews from various computer magazines suggest.

4.2.5 Other Factors

This section presents the remaining factors which may or may not have effected the results of the software engineering project.

- □ **Project Management** :
  For each group, one member was elected as project leader. His/her role was to oversee the software development process. Every member of each group was asked to provide a score (out of 10) on how the project was managed.

- □ **APT Methodology** :
  The students were also asked to provide a score (out of 10) for the usefulness of the APT methodology. This was basically to gain some information on the worth of the methodology, especially in a university environment.

- **Communication With Client** :

  The user requirements for the system were provided by the client via tele-conferencing and written exchanges channelled through the project co-ordinator. The students were asked to award a score (out of 10) on how satisfied they were with the method(s) used for communicating with the client.

- **Team Work** :

  The students were asked to award a score (out of 10) on how satisfied they were with the way their group operated.

- **Contribution To Project** :

  The students were also asked to award a score (out of 10) on how satisfied they were that their contribution was being valued by the rest of the team.

- **On Schedule** :

  The students were asked whether they felt they were able to complete their project on schedule.

Figure 4.2.5.1 presents the results obtained. The results are all based on group averages. **Figure 4.2.5.2** to **Figure 4.2.5.6** presents each of these factors in ascending order.

Group Number	APT Methodology	Communication With Client	Project Management	Team Work	Contribution To Project	On Schedule
1	4.2	3.2	6.6	7.0	8.2	Yes
2	4.2	4.6	7.0	7.0	7.4	No
3	3.5	3.3	5.0	3.8	6.5	Nc
4	4.9	3.2	9.0	8.7	8.8	No
5	5.3	3.0	8.3	7.5	8.3	Yes
6	3.8	2.8	5.7	5.6	7.4	No
7	2.8	3.0	4.3	5.0	5.8	No
8	4.3	4.0	8.3	6.7	8.3	Yes
9	3.0	3.3	8.3	8.8	9.0	No
10	3.4	1.0	7.3	5.5	8.0	No
Average	3.9	3.1	6.9	6.6	7.8	
Minimum	2.8	1.0	4.3	3.8	5.8	
Maximum	5.3	4.6	9.0	8.8	9.0	

Figure 4.2.5.1 **Table - Other Factors That Affect The Project**

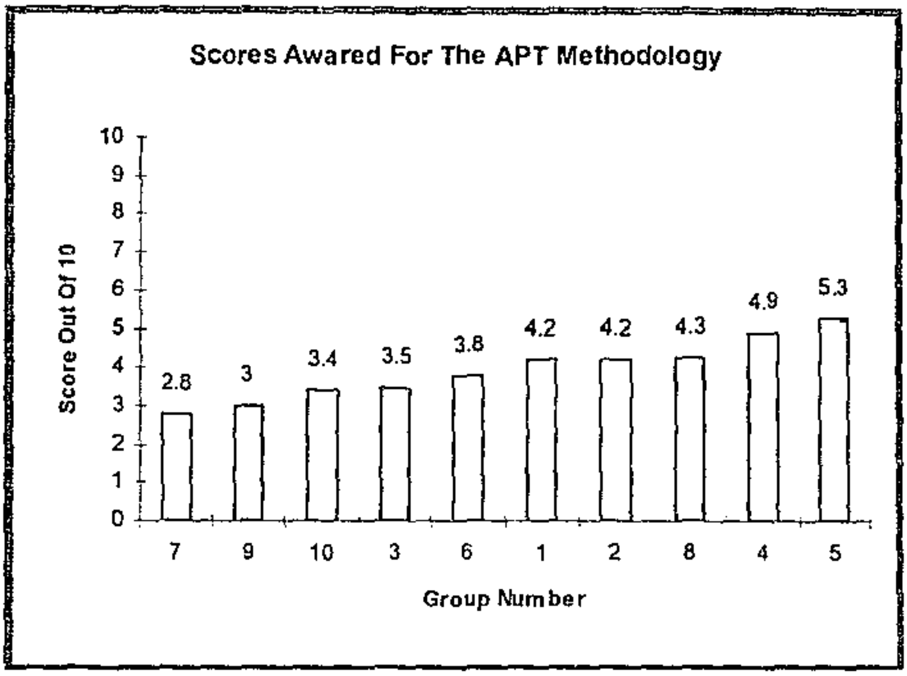


Figure 4.2.5.2 **Graph - Scores Awarded For The APT Methodology**

The APT methodology (EXECOM, 1991) is the standard development methodology adopted by Edith Cowan University's Computer Science Department. To use the APT methodology, students must first purchase the licence for the methodology. Although this methodology has been used by this department since 1991, there was not any information regarding its effectiveness in developing software in a university environment. As a result, 1993's software engineering students were asked on h ›w they felt about this methodology. From the score awarded by the students (see **Figure 4.2.5.1** and **Figure 4.2.5.2**), it can been seen that the APT methodology was not well received. The average score for was 3.9, with a minimum of 2.8 and a maximum of 5.3. As for its usefulness, almost all the students reported that the APT methodology was either not complete (student's version) and/or not suitable for developing software using 4GLs tools and software.

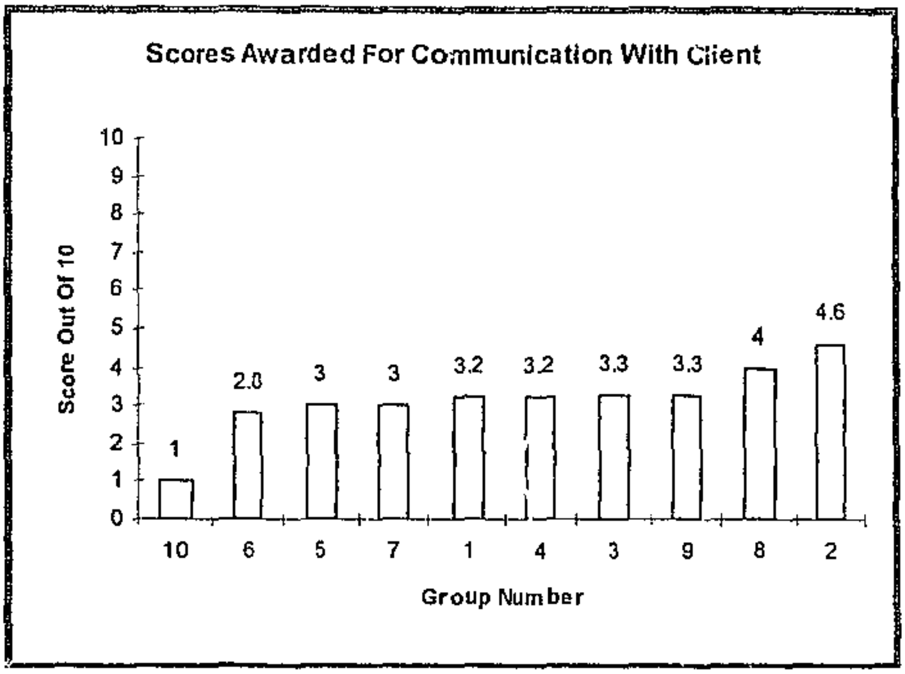


Figure 4.2.5.3 **Graph - Scores Awarded For Communication With Client**

As previously mentioned, the user requirements were provided by the client via tele-conferencing. In total, there were two such conferences, each lasted for about an hour. If the students were to have any questions, they were asked to forward them to their staff adviser or to the project coordinator. Students had no direct access to the client. In theory, information was to be passed between the client and the students via the project coordinator. The scores awarded by the students (see **Figure 4.2.5.1** and **Figure 4.2.5.3**) indicate that this method of communication was not very effective - with an average of 3.1, minimum of 1 and maximum of 4.6. The students had a tight schedule to meet and information was not obtained and provided efficiently. As a result, all the systems that were evaluated addressed different aspects of the orchard business. Some members of the judging panel were heard to remark that, "if some of the various teams' software were combined, it would make a better application". It is fair to say that the user requirements were poorly defined from the students' point of view.

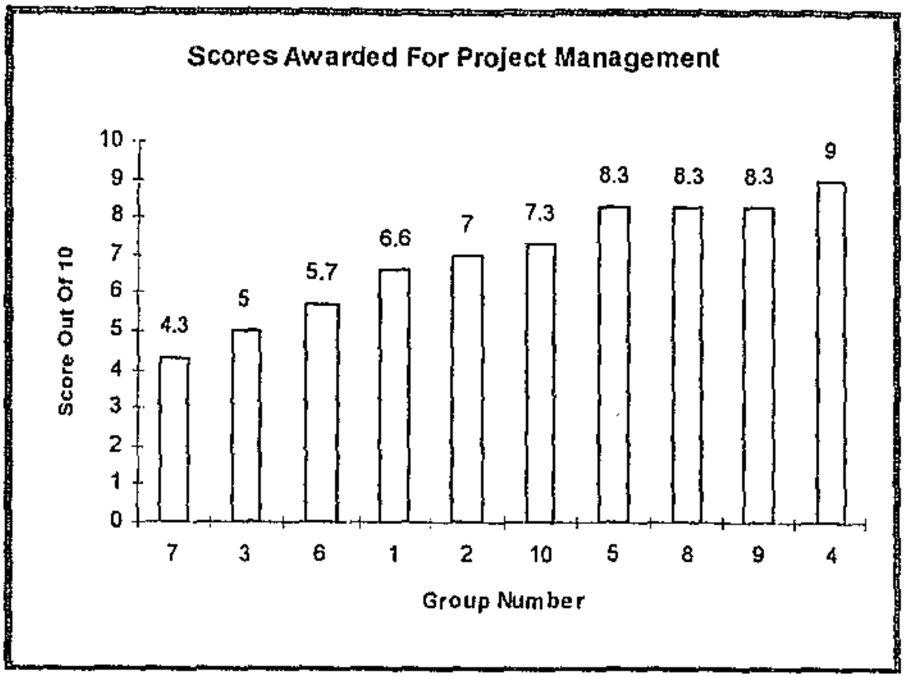


Figure 4.2.5.4 **Graph - Scores Awarded For Project Management**

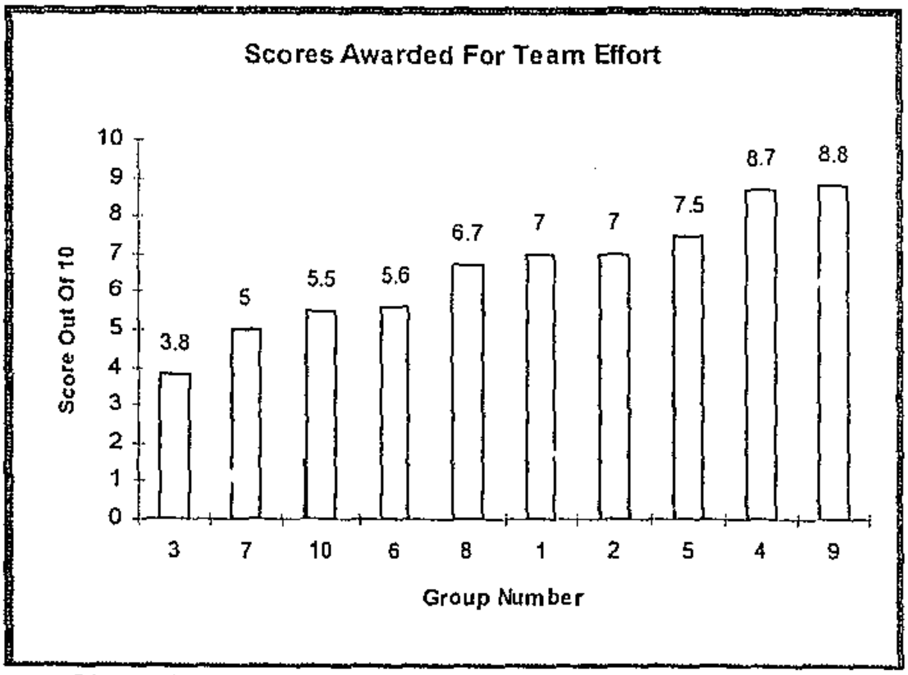


Figure 4.2.5.5 **Graph - Scores Awarded For Team Effort**

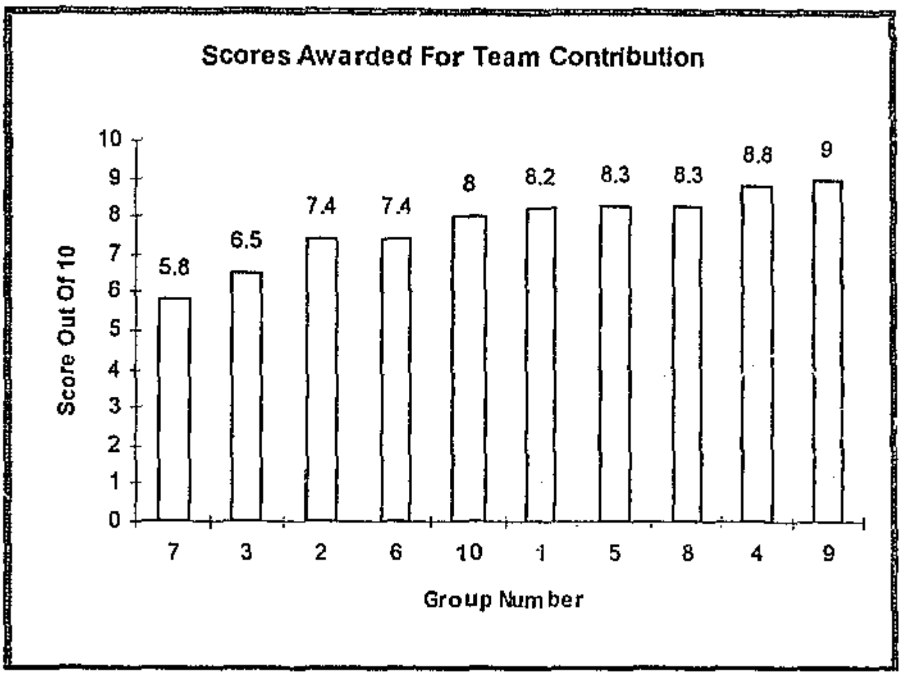


Figure 4.2.5.6 **Graph - Scores Awarded For Team Contribution**

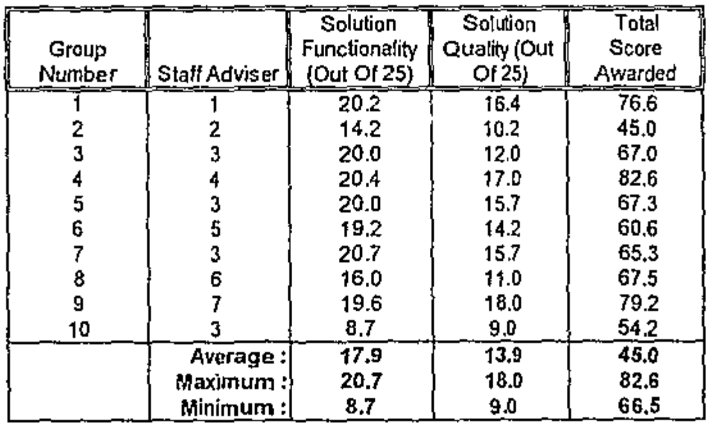
A peer assessment was also conducted. By this, students from each group were requested to give a score (out of 10) on how they felt the project was being managed, how well did the students work as a group and the contributions made by each student to the project. Based on the result from **Figure 4.2.5.1** (also see **Figure 4.2.5.4** to **Figure 4.2.5.6**), the majority of them managed quite well, except for Groups 3 and 7. Group 7 has one of the lowest scores for Project Management, Team Work and Contribution To Project. Whereas Groups 4 and 9 scored extremely well on all counts. It is interesting to note that although these groups had good project management views and high team spirits, it does not automatically follow that their productivity rate will be high (See **Chapter** 7 for details on individual groups' productivity rate).

4.4 <u>SCORE AWARDED TO PROJECTS</u>

This section presents the score that was awarded to each project by the judging panel. Members of the judging panel included the client, project coordinator, unit coordinator and the group's respective staff adviser. Each group questioning towards the end of the demonstration. The scores awarded were based on the groups' :

- Presentation
- Statement of the problem
- Approach to the problem
- Documentation at the presentation
- Solution functionality
- Solution qualitv
- Quality of design for the software

The scores for solution functionality were awarded by the judging panel based on their perception of the solution's functionalities. The scores for solution quality were awarded by the judging panel based on their perception of the quality (usability, fitness for purpose, performance) of these functionalities. These results are presented in **Figures 4.4.1** to **4.4.4**.



Group Number	Staff Adviser	Solution Functionality (Out Of 25)	Solution Quality (Out Of 25)	Total Score Awarded
1	1	20.2	16.4	76.6
2	2	14.2	10.2	45.0
3	3	20.0	12.0	67.0
4	4	20.4	17.0	82.6
5	3	20.0	15.7	67.3
6	5	19.2	14.2	60.6
7	3	20.7	15.7	65.3
8	6	16.0	11.0	67.5
9	7	19.6	18.0	79.2
10	3	8.7	9.0	54.2
	Average :	17.9	13.9	45.0
	Maximum :	20.7	18.0	82.6
	Minimum :	8.7	9.0	66.5

Figure 4.4.1 **Table - Solution Functionality And Solution Quality**

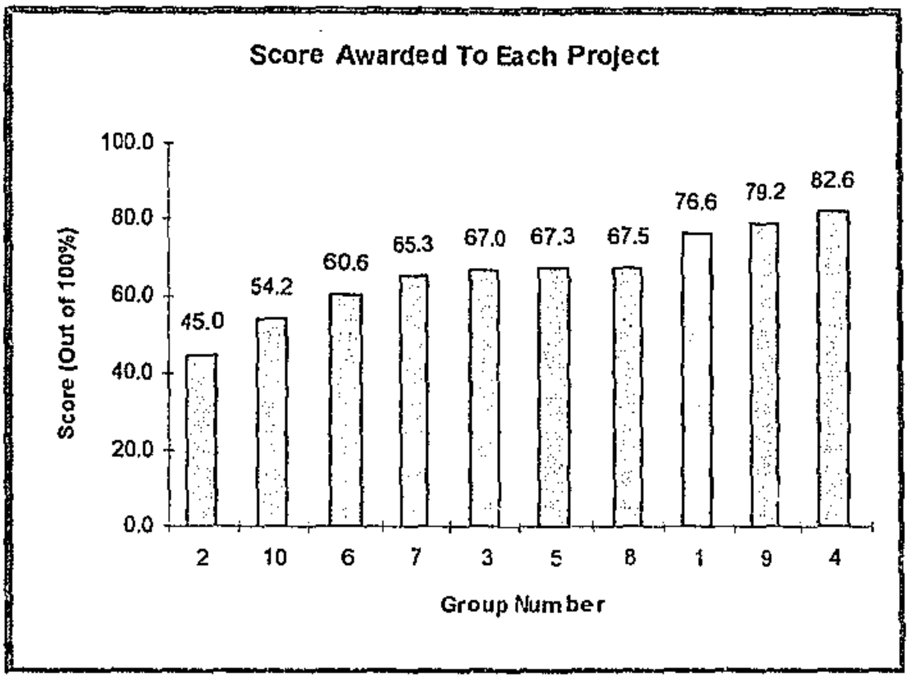


Figure 4.4.2 **Graph - Students' Project Score Sorted In Ascending Order**

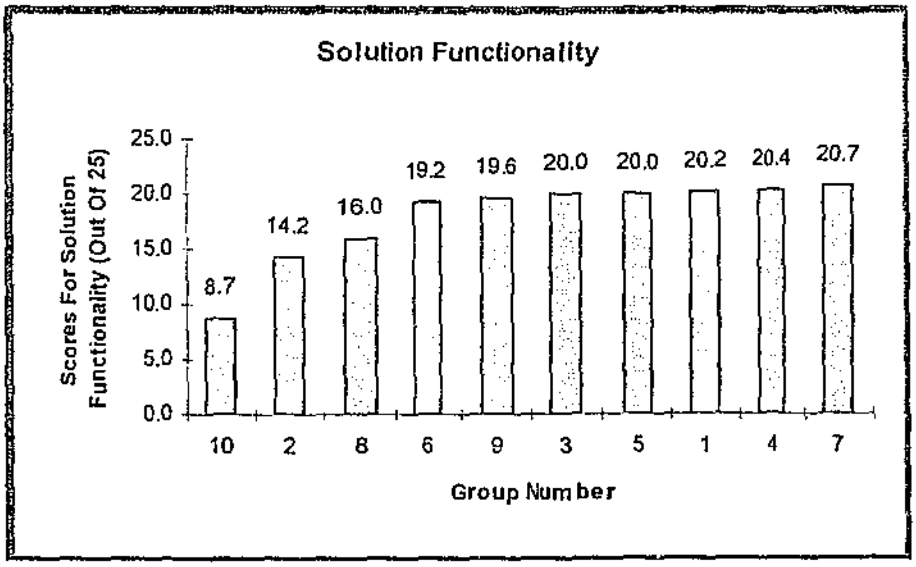


Figure 4.4.3 **Graph - Solution Functionality**

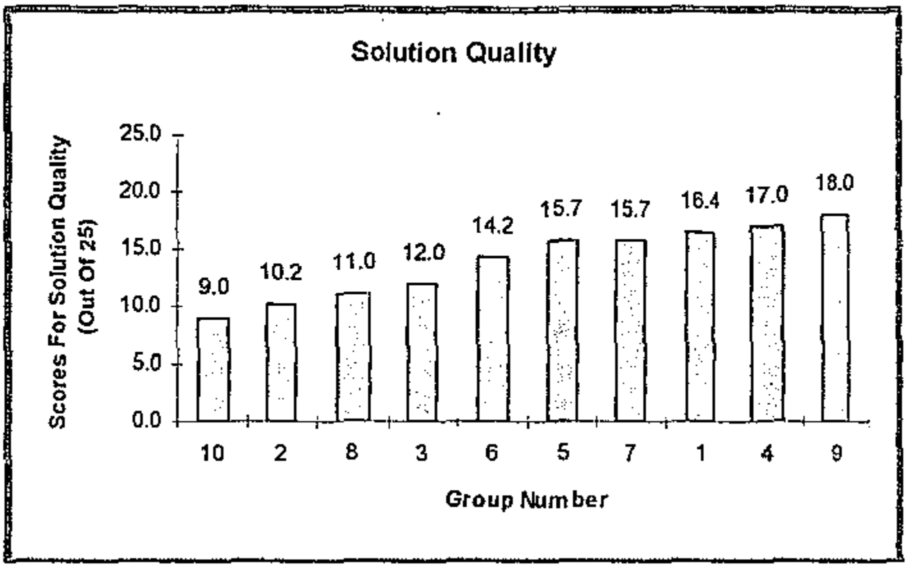


Figure 4.4.4 **Graph - Solution Quality**

4.5 PEER ASSESSMENT SCORES

At the end of each semester, the students of each group were asked to award a mark for their team-mates' performance and contribution to the project. Students were requested to award a score out of 13 and 15 for semester one and two, respectively. These scores are presented in **Figure 4.5.1** and **4.5.2.**

	Peer Assessment		
Group Number	Semester 1 (Out Of 13)	Semester 2 (Out Of 15)	Total (Out Of 28)
1	11.38	11.93	23.31
2	10.15	12.23	22.38
3	10.07	12.08	22.15
4	11.30	13.28	24.58
5	9.82	11.69	21.51
6	11.95	11.29	23.24
7	11.05	13.50	24.55
8	9.17	9.63	18.80
9	13.00	15.00	28.00
10	8.42	9.53	17.95
Average	**10.63**	**12.02**	**22.65**
Minimum	**8.42**	**9.53**	**17.95**
Maximum	**13.00**	**15.00**	**28.00**

Figure 4.5.1 **Table - Total Peer Assessment Scores**

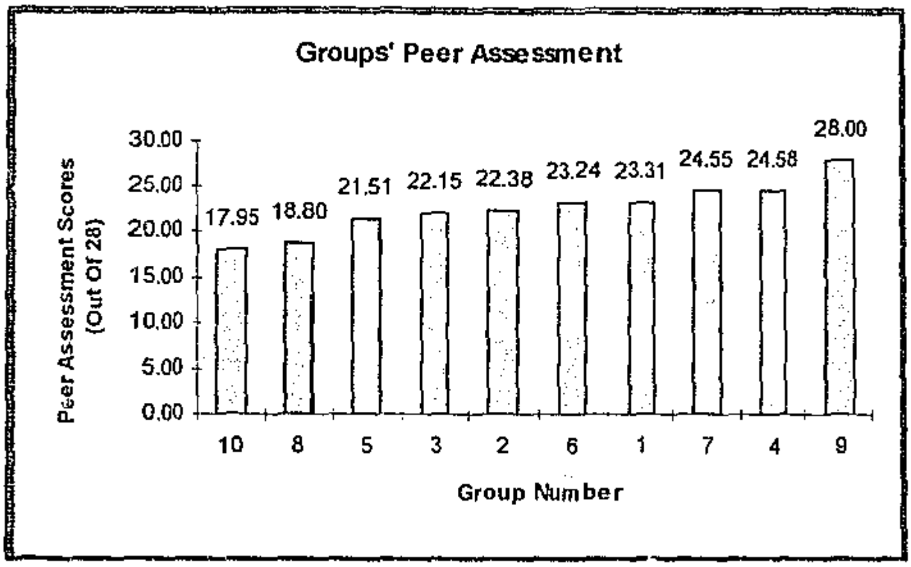


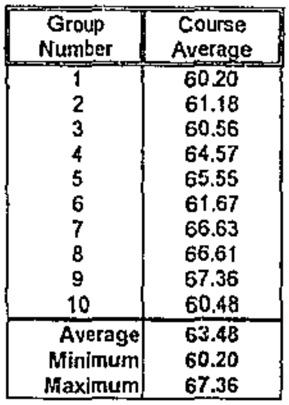
Figure 4.5.2 **Graph - Total Peer Assessment Scores**

From the figures presented above, the majority of the groups did reasonably well. **Group 10** has again received the lowest score. **Group 9** has the largest group and they all scored each other very well. It is obvious that this group worked well as a team. This is supported by the data gathered for Project Management, Team Work and Team Contribution (**For more information, see Section 4.2.5 - Figure 4.2.5.1**).

4.6 GROUPS' COURSE AVERAGES

The formation of groups for the Software Engineering Project was based on the students' course averages. The project coordinator[11] at that time, selected the students for each group based on their individual course averages. The objective was to distribute the students between the groups to provide a reasonable academic balance. The data are presented in **Figures 4.6.1** and **4.6.2**.

[11] Mr Ah Hung, former lecturer and software engineering project coordinator, who has left the employment of this university.



Group Number	Course Average
1	60.20
2	61.18
3	60.56
4	64.57
5	65.55
6	61.67
7	66.63
8	66.61
9	67.36
10	60.48
Average	63.48
Minimum	60.20
Maximum	67.36

Figure 4.6.1 **Table - Groups' Course Averages**

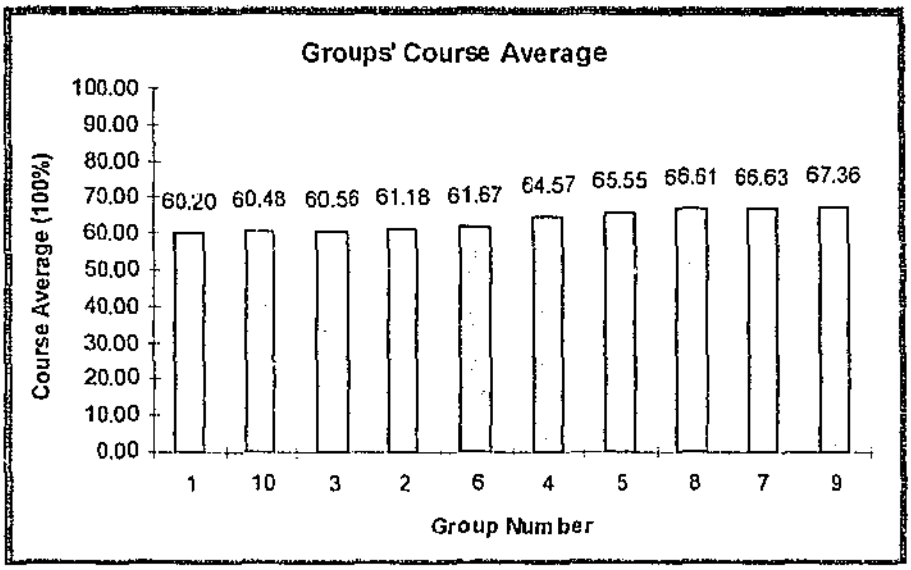


Figure 4.6.2 **Graph - Groups' Course Averages**

Based on the data above, the course averages have a range difference of around 7 per cent.

5.1 SOFTWARE INSTALLABILITY

The Orchard project was to provide students with a real-life problem and the students' task was to develop an application that could be marketable or at least usable by the client. It was expected that the software presented should be as professional as possible. Before the software could be used, it must first be installed on the client's computer. Since not everyone was computer literate, the software installation program (if any) should perform most of the installation process without or with a minimum of user intervention. This section would present the outcome of the investigation into the installability of the software developed by each group.

5.1.1 Software Installation Process

Group 1 : It did not have any installation program. The user needed to create a directory on the hard disk and then copy all the files from the floppy disk over to the hard disk. To execute the software, the user would need to start Windows and then load Microsoft Access version 1.0. From Access, the user could then open the necessary file to execute the application.

Group 2 : There was no installation program. All the associated files were compressed so that it would fit onto one high density floppy disk. Unfortunately, the students failed to provide the software utility for retrieving these compressed files. To retrieve the software, the user would need to create a new directory on the hard disk, then uncompress the file onto the new directory. The procedure to execute this application was identical to that of **Group 1**.

Group 3 : This software came with an installation program in form of a DOS batch file. All the necessary files were compressed into a self-extracting[12] file. All the batch file did was execute this self-extracting file. The user had to be aware of the need to create a new directory on the hard disk first, then to copy all the files on the floppy disk onto this directory. Only then could the user execute the batch file. To execute the application, the user had to first start Windows and then load Paradox for Windows. Once in Paradox for Windows, the user had to then set the Working and Private Directory to the directory where the application's files were located. The user could then start the application by selecting the right form[13].

Group 4 : This was the same situation as **Group 1**.

Group 5 : There was no installation program for this group. The situation was the same as **Group 1** with the exception of a batch start-up file. By running this start-up file, it would automatically load Windows and Microsoft Access, and start the application. Unfortunately, the path for Windows and Access were hard-coded into the batch file and if the user had Windows, Access and the application located in different directories, the batch file would fail in the start-up process.

[12] A self-extracting file was a file that contains all the files that are compressed. It comes in form of an executable file. By executing this file would automatically uncompress all the files.

[13] In this context, a form refers to either an input or output screen, created either by the user or an application generator.

Group 6 : It had no installation program, however all the necessary files were compressed in a self-extracting file. The user would need to create a directory on the hard disk, copy the self-extracting file over to the directory and uncompress the file from there. The execution procedure was identical to that of **Group 3**.

Group 7 : The software had its own installation program. The installation program was created by Microsoft Access Distribution Kit version 1.1. The installation procedure was like any standard installation program found in all Microsoft products. With the disk, came the installation program and the Microsoft Access Runtime module. This is an ideal situation for the user, especially when the software is being distributed for use. The software could still be executed from Access but only in version 1.1.

Group 8 : It had its own installation program. Similar situation as **Group 7** except that the software was developed under Access version 1.0.

Group 9 : This software also had an installation program but it was not created from Microsoft Access Distribution Kit. The installation program would install the software on the hard disk but it did not come with the Runtime module. Hence, the application could only be accessed through Microsoft Access.

Group 10 : There was no installation program. The installation process was identical to **Group 1**.

Group 11 - 16 : Despite intense effort, it was not possible to get these pieces of software working. Hence they have been left out of all metrics gathering.

5.2 SUMMARY

Out of the 10 pieces of software, only three had proper professional installation programs. The remaining seven required a considerable degree of user intervention. For a user who had experience using an operating system such as DOS, this would not pose a problem. However for a user who was not computer literate, it is probable they may have not been capable of installing the software. Unfortunately, access was not provided to either of the user's or technical manuals which means a more detailed assessment into the installability of the proposed software could not be undertaken.

The results of this assessment should take into account the fact that the Department of Computer Science was unable to provide the students with the necessary tools and software. For example, to create an installation program for software developed in Microsoft Access and Paradox for Windows, the Microsoft Access Distributed Kit and Paradox Application Distribution Kit were required. It would appear that the groups that created their own installation program used their own distribution kit and regrettably this resource was not available to all the groups. Access to these resources would definitely enhance the students' learning process, with particular regard to the development of a professional piece of software. It was surprising to find that **Group 10**, which used Objectvision Pro, did not have a good installation program. After all, Objectvision Pro comes with its own Runtime module.

Software that comes with its own Runtime module does not require the client to have a copy of the development software. For example, to execute **Group 7's** application, the installation program will load Access' Runtime module together with the application. From the client's point of view, he or she need not purchase Microsoft Access. From a security point of view, the user will not be able to modify the design of the application directly.

6.1 MEASURING SOFTWARE SIZE USING ALBRECHT'S FUNCTION POINT ANALYSIS

As part of this research project, all the software developed by the 1993 software engineering students was measured to determine the size. The metric used, was Albrecht's Function Points. This metric was selected because it has been widely accepted and used. Furthermore, all the software was produced using 4GL-type development software - Microsoft Access™, Borland Paradox™ for Windows, Borland Objectvision™ Pro and Gupta SQLWindows™. With 4GL-type applications, it is difficult (if not impossible) to determine the size in terms of lines of code of the software because they usually include automated coding. Therefore, it is more reasonable to calculate the size of software in terms of functions delivered rather than lines of code produced.

In total, there were 16 groups of students developing the same application. However, only 15 groups submitted their software for evaluation. All software appeared to function during the project demonstrations[14]. Unfortunately, out of the 15 pieces of software that was submitted for evaluation, only 10 were found to be functioning. Out of the 10 pieces of software, one of them required some modifications before it was capable of being executed on the computer where the evaluation was to be conducted. Of the five pieces of software that were not functioning, one of them was because the students failed to provide a password for their software. With the remaining four pieces of software, it appears that the students failed to submit their final version for evaluation.

[14] The project demonstration was part of the project assessment. Each group was required to demonstrate their software before a judging panel.

Although function points can be counted from the requirements specification the groups' project documentation was not available for this purpose. Furthermore, it was felt that counting function points from the software delivered would yield a more accurate result in relation to the number of function points delivered.

6.2 APPROACH USED TO MEASURE SOFTWARE SIZE

Even though function point analysis is widely discussed in the literature, none provide a detailed description on the procedure involved in counting function points. The primary reason is that the methods available for counting function points are constantly being revised by the International Function Points User's Group (IFPUG). The only published materials that provides an up-to-date description on the procedure for counting function points are published by IFPUG itself. Unfortunately, the latest version of the Function Points Manual was unavailable. As a result, the method for counting function points was taken from Dr. Eberhard Rudolph's[15] (1989) seminar paper. Even though, Dr. Rudolph's paper was slightly dated it proved to be quite useful since the 1993 software engineering project was a straight database-type application running on a standalone computer.

The following sections will explain how the **processing complexity** was defined, how the size of the software was determined and the problems encountered during the evaluation.

[15] Dr. Rudolph presented a three day seminar on Function Point Analysis. His methods for counting function points are also recognised by the Australian Software Metrics Association (1993a).

6.2.1 Rules For Counting Function Points

The first stage of counting function points is to count the ***raw function points***. This is achieved by identifying and classifying the individual functions provided by the software for its end-user. As mentioned in Section **2.4.2 Counting Function Points**, there are five types of functions - external input, external output, logical files, external interfaces and external inquiries.

- □ **External input**

  Any data that enters the information system from the user should be considered as an external input. It will be counted when the system adds, changes or deletes data in a logical file type. Therefore, functions that were counted include :

  - data input screen
  - data update screen
  - data deletion screen

- □ **External output**

  An external output type does not modify the contents of the internal logical files. External output types can reach the users directly as reports or messages. External output types of the same format but of different output medium should only be counted as one output type. However, the same information presented in different format, allowing for the characteristics of the output device are counted as separate external outputs. The functions that were counted include :

  - □ Reports
  - □ Start screen output
  - □ End screen output

- **Logical files**

  Each major logical group of user data in the application system should be considered as an internal logical file type. In order to be counted as a logical file, a logical user view had to be generated, used and maintained by the information system. An internal logical file should be directly used by at least one external input, external output or external inquiry type. Internal logical files that are not accessed by an input, output or inquiry types are not to be counted.

- **External interfaces**

  Files or control information that are passed or shared among different systems should be counted within each information system as an external interface type. With the 1993 software engineering project, there were not any external interfaces. However, the client did express interest in the ability to share data between Quicken™ for Windows and the proposed system. Unfortunately, none of the groups were able to achieve this functionality. Therefore, in this case the external interface count was set to zero (0).

- **External inquiries**

  An external inquiry type is a query facility that is offered by the application. It is characterised by a unique input/output combination. It triggers off an immediate response without updating the internal logical files. It is entered to direct the search so that the desired information can be found. The functions that were counted include :

  - Help screens
  - Menu selection screens
  - Lookup tables
  - Online query

6.2.2 Defining The Complexity Adjustment Values

Before deriving the final function points for a piece of software a prerequisite is the determination of the software's processing complexity. This can be determined by adjusting the 14 general application attributes. For each of these attributes, a value must be assigned (degree of influence - **DI**) which ranges from 0 to 5 - where 0 suggests "either not present or no degree of influence" and 5 suggests "strong influence throughout the application development". The following is a list of the 14 attributes with its associated value of influence. The reason for selecting the value of influence for each attribute, is also explained.

- ❑ **Data communication**
  This attribute is present when information is being sent and received over some form of communication facility. This was set to zero (0) because the software was developed for a standalone environment. There was no use of communication facilities such as telephone lines.

- ❑ **Distributed functions**
  This attribute is present when the system's data is distributed and processed over more than one processor. This was set to zero (0) because there was no need for distributed processing. Since the application was developed for a standalone environment, all data were stored and processed locally.

- ❑ **Performance**
  This attribute is present when performance objectives such as response time and throughput are stated and approved by the end user. This was set to two (2) because the performance of the system could be met by standard design and coding practices. The end user had not specifically set the criteria for acceptable performance.

- □ **Heavily used configuration**
  This attribute is present when the system requires special design and implementation considerations. It is typically concerned with main storage or disk storage limitations and processor time. This was set to three (3) because the operational restrictions required minor attention in the project plan.

- □ **Transaction rate**
  This attribute represents the flow of information within the system. This was set to two (2) because the transaction rate was moderate, however this transaction rate could be met with standard design and coding techniques.

- □ **Online data entry**
  This attribute represents the amount of transactions that were entered interactively. This was set to five (5) because more than 30 per cent (in fact, all) of the transactions were entered interactively.

- □ **End user efficiency**
  This attribute gives credit to the emphasis in designing functions that provide efficient user information access. This was set to five (5) because special tools such as 4GLs were used in the design and development phases to promote end-user efficiency.

- □ **Online update**
  This attribute determine the degree of online updates performed by the system. This was set to three (3) because online updating was provided for all the major logical internal files.

- □ **Complex processing**
  This attribute reflects the complexity of the programming logic. This was set to one (1) because of its extensive logical processing.

- □ **Reusability**
  This attribute is present when the code of the resulting application programs has been designed, developed, and supported to be usable in other information systems. This was set to zero (0) because no consideration for reusability was specified.

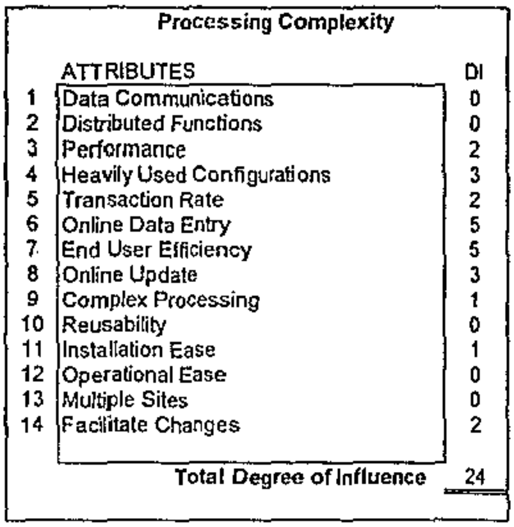
- □ **Installation ease**
  This attribute is present when the information system requires specific installation considerations during its transition from the current system to the new system. This was set to one (1) because a conversion plan was required but no data conversion was needed.

- □ **Operational ease**
  This attribute is present when the system requires effective start-up, back-up and recovery procedures. This was set to zero (0) because no special operational considerations were stated by the user.

- □ **Multiple sites**
  This attribute is present when the system has been specifically designed, developed and supported, to be installed at multiple locations. This was set to zero (0) because there was no requirement to consider more than one location.

- □ **Facilitate change**
  This attribute is present when the system has been designed, developed, and supported to facilitate modifications of its functions at a later stage. This was set to two (2) because the application was to be implemented as a series of modules.

The settings of all these attributes were applied to the ten pieces of software that were evaluated. This was to ensure consistency in the method of measuring function points. Below is the table (Figure 6.2.1.1) representing these attributes with its associated value of influence and the **Total Degree of Influence** that was used to calculate the final function points.



Processing Complexity

	ATTRIBUTES	DI
1	Data Communications	0
2	Distributed Functions	0
3	Performance	2
4	Heavily Used Configurations	3
5	Transaction Rate	2
6	Online Data Entry	5
7	End User Efficiency	5
8	Online Update	3
9	Complex Processing	1
10	Reusability	0
11	Installation Ease	1
12	Operational Ease	0
13	Multiple Sites	0
14	Facilitate Changes	2
	Total Degree of Influence	24

Figure 6.2.2.1 **Table - Processing Complexity Used For Calculating Software Size**

6.2.3 Evaluation Report

All the software that was tested had some form of bugs or logical errors. In some cases, the software caused the system software and Microsoft Windows to crash. See **Appendix A** for the list of errors.

6.2.4 Size Of The Software

This section presents the size of the ten software projects that were evaluated. For ethical reasons, the students' name and group number will remain anonymous. Each group has been assigned a different group number.

The ***Total Unadjusted Function Points*** is derived by adding the totals of the five function types. The ***Adjustment Factor*** is calculated from the equation:

$$\text{Adjustment Factor} = 0.65 + (0.01 \times \text{Total Degree Of Influence})$$

where the ***Total Degree Of Influence*** is obtained from **Figure 6.2.2.1**. The actual function points were calculated using the following equation :

Function Points = Total Unadjusted Function Points × Adjustment Factor

For more information, see **Section 2.4.2 Counting Function Points.**

□ Group 1

External Inputs		36	Simple	× 3	=	108
		6	Average	× 4	=	24
		0	Complex	× 6	=	0
	Totals	42				132
External Outputs		7	Simple	× 4	=	28
		0	Average	× 5	=	0
		0	Complex	× 7	=	0
	Totals	7				28
Logical Files		17	Simple	× 7	=	119
		1	Average	× 10	=	10
		0	Complex	× 15	=	0
	Totals	18				129
Interfaces		0	Simple	× 5	=	0
		0	Average	× 7	=	0
		0	Complex	× 10	=	0
	Totals	0				0
External Inquires		30	Simple	× 3	=	90
		1	Average	× 4	=	4
		0	Complex	× 6	=	0
	Totals	31				94
	Total Unadjusted Function Points				=	383
	Adjustment Factor				:	0.89
	Function Points				:	341

□ Group 2

External Inputs		15	Simple	× 3	=	45
		0	Average	× 4	=	0
		0	Complex	× 6	=	0
	Totals	15				45
External Outputs		22	Simple	× 4	=	88
		0	Average	× 5	=	0
		0	Complex	× 7	=	0
	Totals	22				88
Logical Files		22	Simple	× 7	=	154
		0	Average	× 10	=	0
		0	Complex	× 15	=	0
	Totals	22				154
Interfaces		0	Simple	× 5	=	0
		0	Average	× 7	=	0
		0	Complex	× 10	=	0
	Totals	0				0
External Inquires		6	Simple	× 3	=	18
		0	Average	× 4	=	0
		0	Complex	× 6	=	0
	Totals	6				18
	Total Unadjusted Function Points				=	305
	Adjustment Factor				:	0.89
	Function Points				:	271

□ Group 3

External Inputs		25	Simple	×	3	=	75
		12	Average	×	4	=	48
		3	Complex	×	6	=	18
	Totals	40					141
External Outputs		12	Simple	×	4	=	48
		1	Average	×	5	=	5
		0	Complex	×	7	=	0
	Totals	13					53
Logical Files		29	Simple	×	7	=	203
		0	Average	×	10	=	0
		0	Complex	×	15	=	0
	Totals	29					203
Interfaces		0	Simple	×	5	=	0
		0	Average	×	7	=	0
		0	Complex	×	10	=	0
	Totals	0					0
External Inquires		8	Simple	×	3	=	24
		0	Average	×	4	=	0
		0	Complex	×	6	=	0
	Totals	8					24
	Total Unadjusted Function Points					=	421
	Adjustment Factor					:	0.89
	Function Points					:	375

□ Group 4

External Inputs		37	Simple	×	3	=	111
		3	Average	×	4	=	12
		2	Complex	×	6	=	12
	Totals	42					135
External Outputs		23	Simple	×	4	=	92
		2	Average	×	5	=	10
		2	Complex	×	7	=	14
	Totals	27					116
Logical Files		16	Simple	×	7	=	112
		0	Average	×	10	=	0
		0	Complex	×	15	=	0
	Totals	16					112
Interfaces		0	Simple	×	5	=	0
		0	Average	×	7	=	0
		0	Complex	×	10	=	0
	Totals	0					0
External Inquires		11	Simple	×	3	=	33
		1	Average	×	4	=	4
		0	Complex	×	6	=	0
	Totals	12					37
	Total Unadjusted Function Points					=	400
	Adjustment Factor					:	0.89
	Function Points					:	356

❑ Group 5

External Inputs		24	Simple	×	3	=	72
		3	Average	×	4	=	12
		15	Complex	×	6	=	90
	Totals	42					174
Extern l Outputs		1	Simple	×	4	=	4
		0	Average	×	5	=	0
		6	Complex	×	7	=	42
	Totals	7					46
Logical Files		16	Simple	×	7	=	112
		0	Average	×	10	=	0
		0	Complex	×	15	=	0
	Totals	16					112
Interfaces		0	Simple	×	5	=	0
		0	Average	×	7	=	0
		0	Complex	×	10	=	0
	Totals	0					0
External Inquires		3	Simple	×	3	=	9
		2	Average	×	4	=	8
		0	Complex	×	6	=	0
	Totals	5					17
	Total Unadjusted Function Points					=	349
	Adjustment Factor					:	0.89
	Function Points					:	311

❑ Group 6

External Inputs		42	Simple	×	3	=	126
		6	Average	×	4	=	24
		1	Complex	×	6	=	6
	Totals	49					156
External Outputs		0	Simple	×	4	=	0
		0	Average	×	5	=	0
		0	Complex	×	7	=	0
	Totals	0					0
Logical Files		36	Simple	×	7	=	252
		0	Average	×	10	=	0
		0	Complex	×	15	=	0
	Totals	36					252
Interfaces		0	Simple	×	5	=	0
		0	Average	×	7	=	0
		0	Complex	×	10	=	0
	Totals	0					0
External Inquires		19	Simple	•	3	=	57
		0	Average	•	4	=	0
		0	Complex	•	6	=	0
	Totals	19					57
	Total Unadjusted Function Points					=	465
	Adjustment Factor					:	0.89
	Function Points					:	414

❑ Group 7

External Inputs		18	Simple	×	3	=	54
		9	Average	×	4	=	36
		6	Complex	×	6	=	36
	Totals	33					126
External Outputs		3	Simple	×	4	=	12
		1	Average	×	5	=	5
		4	Complex	×	7	=	28
	Totals	8					45
Logical Files		20	Simple	×	7	=	140
		2	Average	×	10	=	20
		0	Complex	×	15	=	0
	Totals	22					160
Interfaces		0	Simple	×	5	=	0
		0	Average	×	7	=	0
		0	Complex	×	10	=	0
	Totals	0					0
External Inquires		4	Simple	×	3	=	12
		0	Average	×	4	=	0
		1	Complex	×	6	=	6
	Totals	5					18
	Total Unadjusted Function Points					=	349
			Adjustment Factor			:	0.89
			Function Points			:	311

❑ Group 8

External Inputs		18	Simple	×	3	=	54
		0	Average	×	4	=	0
		0	Complex	×	6	=	0
	Totals	18					54
External Outputs		5	Simple	×	4	=	20
		0	Average	×	5	=	0
		0	Complex	×	7	=	0
	Totals	5					20
Logical Files		9	Simple	×	7	=	63
		0	Average	×	10	=	0
		0	Complex	×	15	=	0
	Totals	9					63
Interfaces		0	Simple	×	5	=	0
		0	Average	×	7	=	0
		0	Complex	×	10	=	0
	Totals	0					0
External Inquires		8	Simple	×	3	=	24
		0	Average	×	4	=	0
		0	Complex	×	6	=	0
	Totals	8					24
	Total Unadjusted Function Points					=	161
			Adjustment Factor			:	0.89
			Function Points			:	143

□ Group 9

External Inputs	16	Simple	×	3	=	48
	0	Average	×	4	=	0
	1	Complex	×	6	=	6
Totals	17					54
External Outputs	4	Simple	×	4	=	16
	3	Average	×	5	=	15
	0	Complex	×	7	=	0
Totals	7					31
Logical Files	14	Simple	×	7	=	98
	0	Average	×	10	=	0
	0	Complex	×	15	=	0
Totals	14					98
Interfaces	0	Simple	×	5	=	0
	0	Average	×	7	=	0
	0	Complex	×	10	=	0
Totals	0					0
External Inquires	8	Simple	×	3	=	24
	0	Average	×	4	=	0
	2	Complex	×	6	=	12
Totals	10					36
Total Unadjusted Function Points					=	219
Adjustment Factor					:	0.89
Function Points					:	195

□ Group 10

External Inputs	3	Simple	×	3	=	9
	0	Average	×	4	=	0
	0	Complex	×	6	=	0
Totals	3					9
External Outputs	0	Simple	×	4	=	0
	0	Average	×	5	=	0
	0	Complex	×	7	=	0
Totals	0					0
Logical Files	12	Simple	×	7	=	84
	0	Average	×	10	=	0
	0	Complex	×	15	=	0
Totals	12					84
Interfaces	0	Simple	×	5	=	0
	0	Average	×	7	=	0
	0	Complex	×	10	=	0
Totals	0					0
External Inquires	3	Simple	×	3	=	9
	0	Average	×	4	=	0
	0	Complex	×	6	=	0
Totals	3					9
Total Unadjusted Function Points					=	102
Adjustment Factor					:	0.89
Function Points					:	91

6.2.5 Scores Awarded For Solution Functionality

This section presents the score awarded by the judging panel for the groups' software solution functionality and presentation skills. These scores were awarded by the judging panel based on a presentation given by each group. The score is given out of 25 points. The results are presented in **Figure 6.2.5.1** and **Figure 6.2.5.2.**

Group Number	Size Of Software In Function Points	Solution Functionality (Score Out Of 25)
1	341	20.2
2	271	14.2
3	375	20
4	356	20.4
5	311	20
6	414	19.2
7	311	20.7
8	143	16
9	195	19.6
10	91	8.7
Average :	280.8	17.9
Minimum :	91	8.7
Maximum :	414	20.7

Figure 6.2.5.1 **Table - Scores Awarded For Solution Functionality**

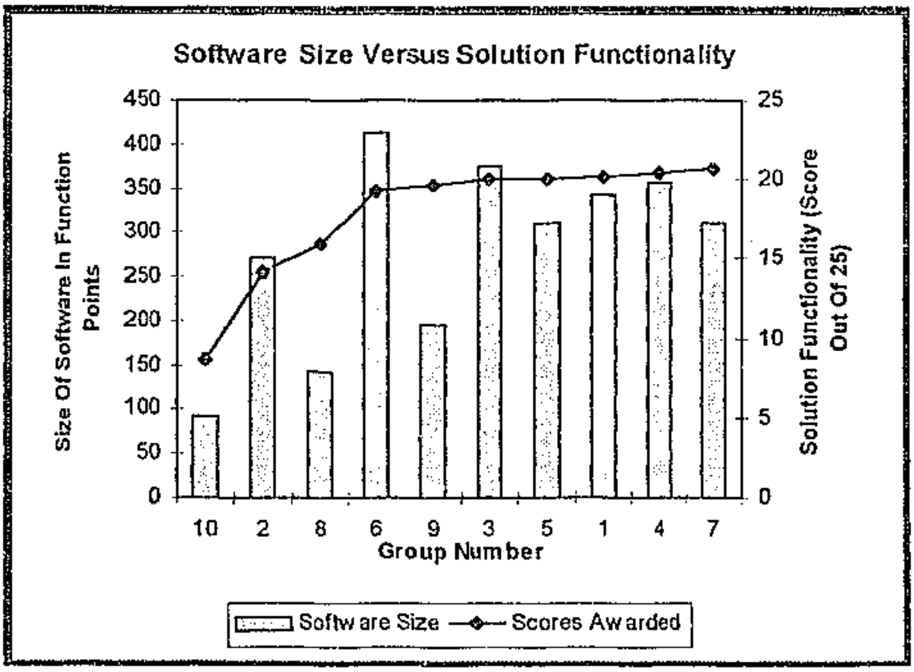


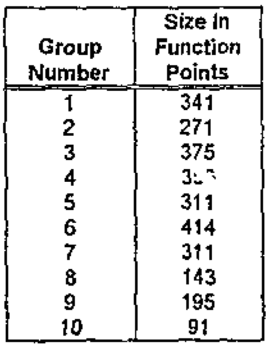
Figure 6.2.5.2 **Graph - Software Size Versus Solution Functionality (Sorted According To Score)**

Based on the results presented above, the majority of the software with a size of 300+ function points scored around 20 points, except for Group 6. This group had the highest function point count of 414 but only scored 19.2 points. The results did however reflect the fact that software with poor functionality scored less. For example, Group 10 had the smallest size of 91 function points and it only scored 8.7 points (the lowest).

6.2.6 Summary

This chapter presents the size of the ten pieces of software that were evaluated using Albrecht's Function Point method. Even though these pieces of software were quite functional, they were in no way near "perfect" or ready to be used by the client. For each piece of software that was tested, a brief "evaluating report" was presented to indicate the functionality of the software. By doing so, it provides a comparison of the software's functionality against its size.

Figure 6.2.6.1 and **Figure 6.2.6.2** presents the size of software for each group in function points. **Figure 6.2.6.3** and **Figure 6.2.6.4** presents the size of each function type for each group, in terms of function points.



Group Number	Size In Function Points
1	341
2	271
3	375
4	[illegible]
5	311
6	414
7	311
8	143
9	195
10	91

Figure 6.2.6.1 **Table - Size Of Software Per Group**

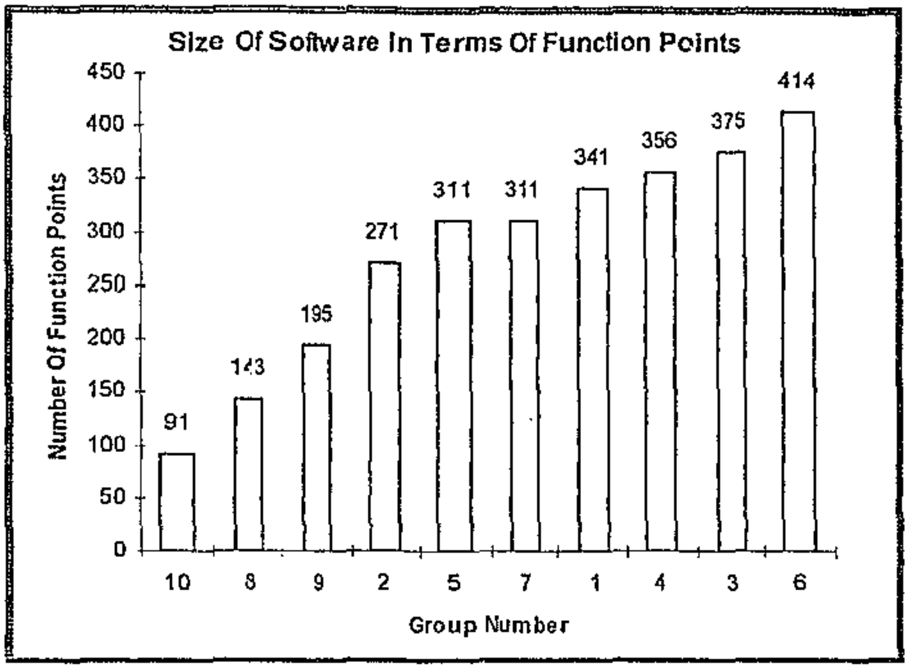


Figure 6.2.6.2 **Graph - Size Of Software (Sorted In Ascending Order)**

By looking at the graph in **Figure 6.2.6.2** and the data presented in the evaluation report (**Appendix A**), it can be seen that the largest software is not necessarily seen as the most "functional". Although in theory, it could be expected that the software with higher function point count would indeed be more "functional". Even though Group 6 had the largest function count, comments presented in their evaluation report indicate that their software was not well developed. In fact, it was one of two that crashed not only the system software environment, but it also crashed the operating environment (Microsoft Windows). Group 3's software also crashed the operating environment when trying to access one of its modules, yet this group has the second highest function point count.

Looking at the two extreme ends, Groups 10 and 8 rank the smallest in size. Again, by examining the evaluation report, it can be seen that both pieces of software lack in functionality. Group 10's software did not have reports and most of its functions were poorly developed. Group 8's software was relatively easy to use but it lacked in functionality.

Group Number	External Input	External Output	Logical Files	External Inquires
1	132	28	129	94
2	45	88	154	18
3	141	53	203	24
4	135	116	112	37
5	174	46	112	17
6	156	0	252	57
7	126	45	160	18
8	54	20	63	24
9	54	31	98	36
10	9	0	84	9

Figure 6.2.6.3 **Table - Size Of Each Function Types**

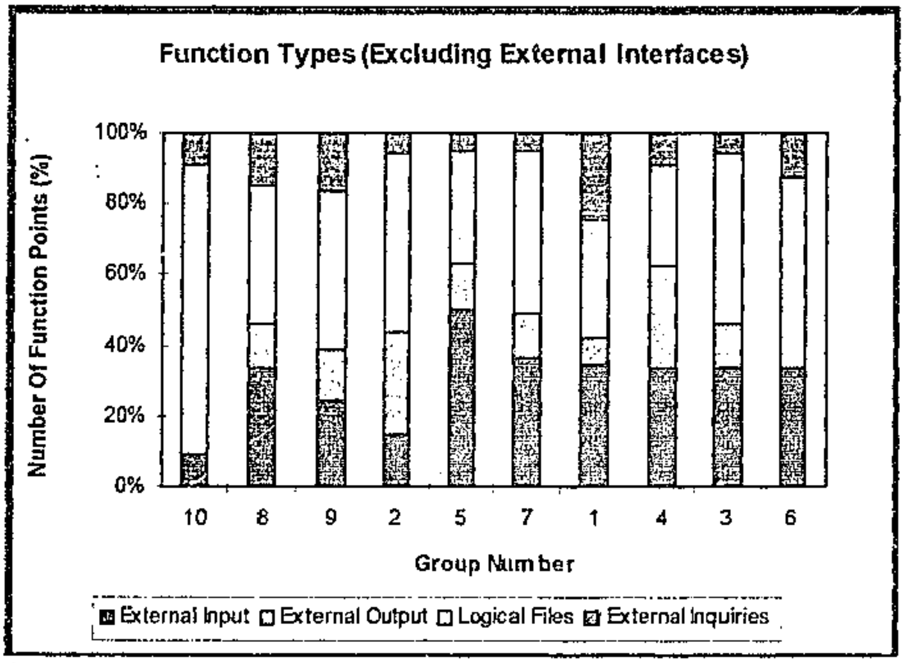


Figure 6.2.6.4 **Graph - Size Of Each Function Types (In Percentage) Sorted According To Overall Size**

From **Figure 6.2.6.4**, both pieces of software with the smallest (Group 10) and largest (Group 6) size had zero (0) for its external output. In Group 10's case, report options were in the menu's structure but they were not functioning when tested. With Group 6, the report options had been completely omitted.

CHAPTER 7 : MEASURING PRODUCTIVITY

7.1 MEASURING THE PRODUCTIVITY OF PROJECT GROUPS

The productivity of developers is mainly concerned with software project management. It is used to measure the software development "output" as a function of effort applied. This chapter will present the productivity of each group based on two methods. The first method is based on the Australian Software Metrics Association Project Databases - Release 3 (1993b). It measures productivity as Project Delivery Rate (ie. the number of hours required to deliver one function point) and is derived using the equation below.

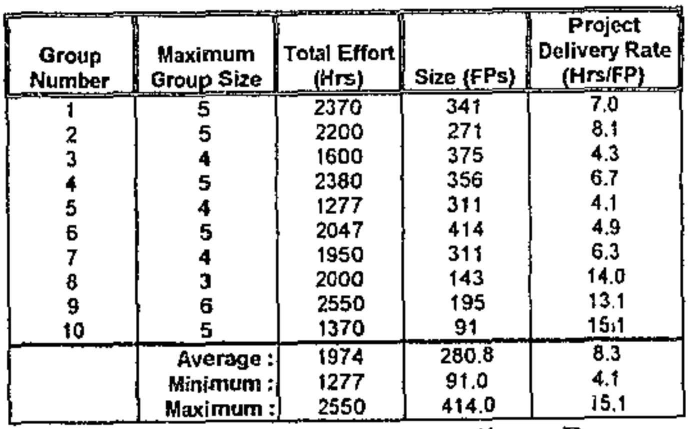
$$\text{Project Delivery Rate} = \frac{\text{Effort (Hours)}}{\text{Size (Function Points)}}$$

The second method is based on the productivity equation that is often used in Function Point Analysis (Pressman, 1992).

$$\text{Productivity} = \frac{\text{Size (Function Points)}}{\text{Person - Month}}$$

7.2 PROJECT DELIVERY RATE

As mentioned above, this method of measuring productivity is based on the documentation provided by the Australian Software Metrics Association Project Database - Release 3 (1993b). **Figure 7.2.1** and **7.2.2** presents the project delivery rate for each group.



Group Number	Maximum Group Size	Total Effort (Hrs)	Size (FPs)	Project Delivery Rate (Hrs/FP)
1	5	2370	341	7.0
2	5	2200	271	8.1
3	4	1600	375	4.3
4	5	2380	356	6.7
5	4	1277	311	4.1
6	5	2047	414	4.9
7	4	1950	311	6.3
8	3	2000	143	14.0
9	6	2550	195	13.1
10	5	1370	91	15.1
	Average :	1974	280.8	8.3
	Minimum :	1277	91.0	4.1
	Maximum :	2550	414.0	15.1

Figure 7.2.1 **Table - Project Delivery Rate**

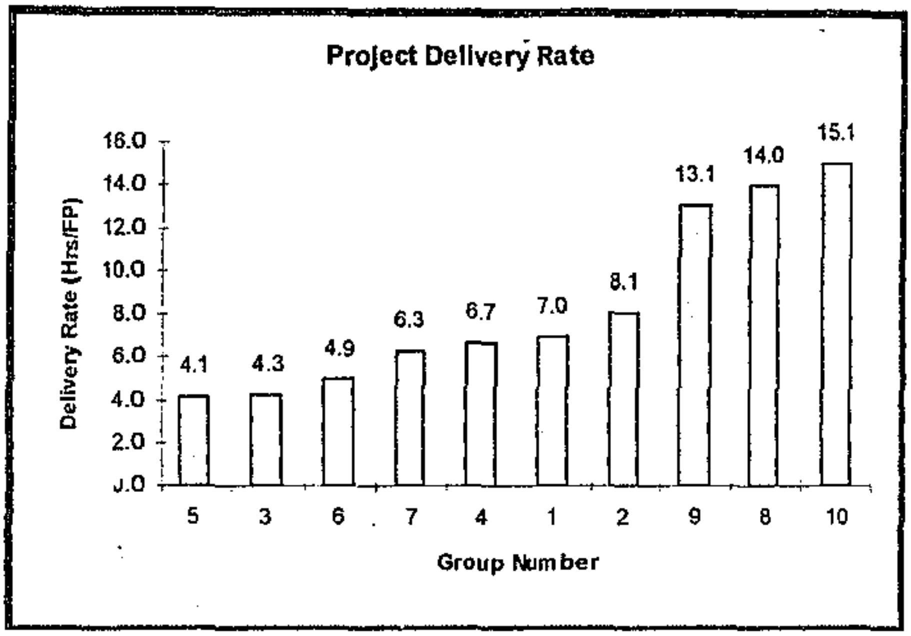


Figure 7.2.2 **Graph - Project Delivery Rate In Ascending Order**

Figure 7.2.3 presents the types of software and hardware development platform used by each group. Seventy per cent of the groups used Microsoft Access, twenty per cent used Paradox for Windows and ten per cent used Objectvision Pro.

Group Number	Language Type	Language Name	Hardware Platform
1	4GL	Microsoft Access	Personal Computer
2	4GL	Microsoft Access	Personal Computer
3	4GL	Paradox For Windows	Personal Computer
4	4GL	Microsoft Access	Personal Computer
5	4GL	Microsoft Access	Personal Computer
6	4GL	Paradox For Windows	Personal Computer
7	4GL	Microsoft Access	Personal Computer
8	4GL	Microsoft Access	Personal Computer
9	4GL	Microsoft Access	Personal Computer
10	4GL	Objectvision Pro	Personal Computer

Figure 7.2.3 **Table - Software Development Platform**

		Breakdown Of Project Delivery Rate		
	Number Of Groups	Minimum	Maximum	Average
Microsoft Access	7	4.1	14.0	8.5
Paradox For Windows	2	4.3	4.9	4.6
Objectvision Pro	1	15.1	15.1	15.1

Figure 7.2.4 **Table - Breakdown Of Delivery Rate By Software Type**

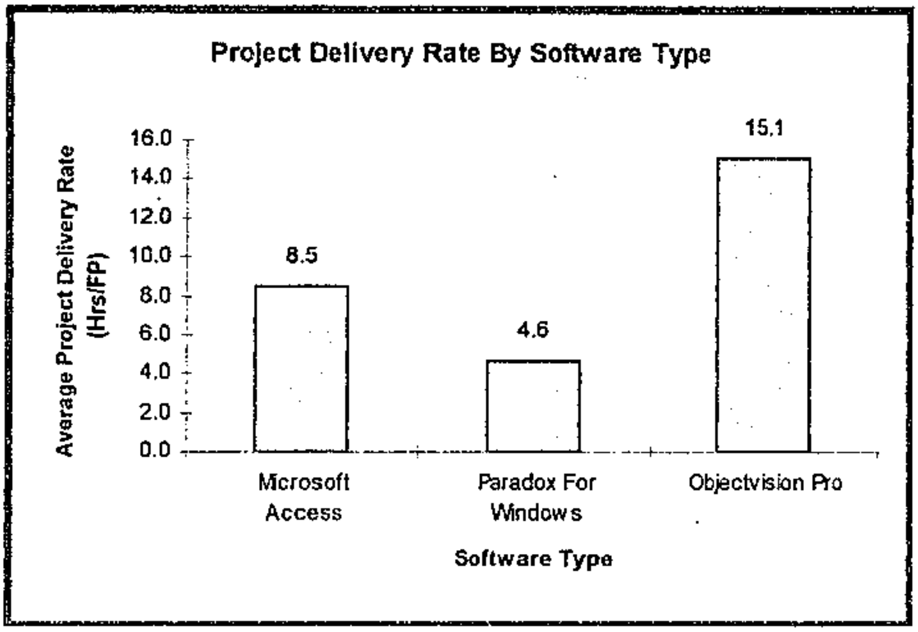


Figure 7.2.5 **Graph - Project Delivery Rate By Software Type**

From the available systems software, the majority of the groups selected Microsoft Access. Surprisingly, groups using Access seem to have a lower delivery rate of 8.5 Hrs/FP than the groups using Paradox for Windows which not only produced the largest software but also had a very high delivery rate, with an average of 4.6 Hrs/FP. It is important to point out that only two groups used Paradox for Windows - a very small sample. Objectvision Pro was only used by one group, therefore it is very difficult to make any definite conclusion.

7.3 PRODUCTIVITY

This section will present the productivity rate of each project group. The method for determining the groups' productivity is derived by dividing the size of the software in function points with the number of person-months worked. Before the productivity rate can be derived, it was necessary to first determine the number of person-months spent developing the software in each group. In this particular case, the number of person-months was defined based on the following three assumptions.

- □ Members of each group spent four hours during the weekdays, working on the project.
- □ Members of each group spent six hours during the weekends, working on the project.
- □ There are four weeks in a month.

This was necessary because the data regarding the number of actual hours spent by each student were not available. It also provides a means to compare the 1993 software engineering project with future students' projects. Based on these assumptions, it is calculated that each student could spend 32 hours per week on the project, bringing a total of 128 hours per month. From this value, the number of person-months spent by each group can be derived. The data are presented in **Figure 7.3.1** and **7.3.2**.

Group Number	Total Effort (Hours)	Person-Month
1	2370	18.5
2	2200	17.2
3	1600	12.5
4	2380	18.6
5	1277	10.0
6	2047	16.0
7	1950	15.2
8	2000	15.6
9	2550	19.9
10	1370	10.7
	Average :	**15.4**
	Minimum :	**10.0**
	Maximum :	**19.9**

Figure 7.3.1 **Table - Deriving Number Of Person-Month**

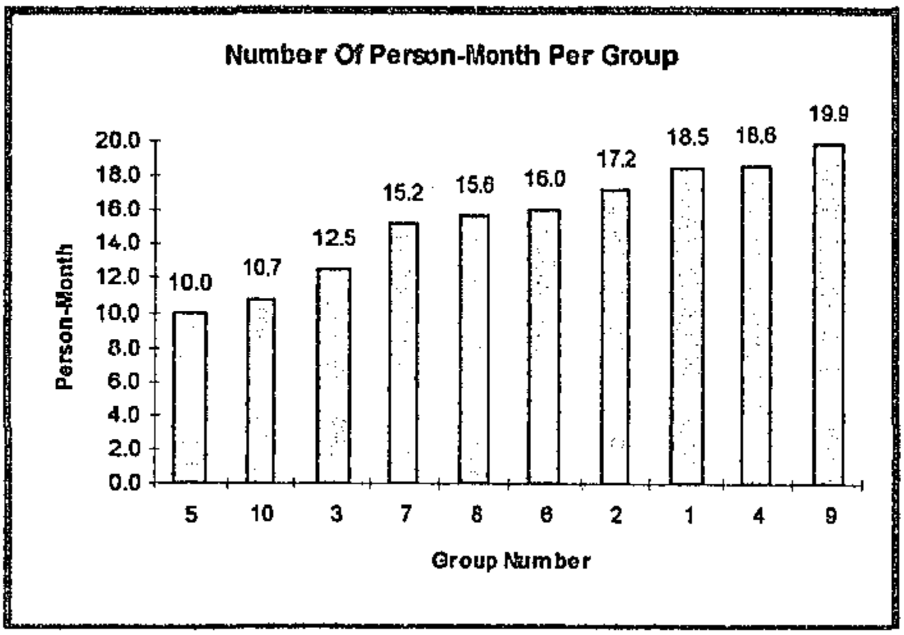


Figure 7.3.2 **Graph - Number Of Person-Month (In Ascending Order)**

After deriving the number of person-months for each group, the productivity rate for each group was determined. **Figures 7.3.3** to **7.3.4** present the productivity rate in terms of function points delivered by each group and by each student (on average) of a group. These figures are derived by using the following the equations.

$$\text{Productivity Rate Per Group - Person} = \frac{\text{Size (Function Points)}}{\text{Number Of Person - Month}}$$

Group Number	Number Of Students	Size (FPs)	Person-Month (PM)	Productivity Rate Per Group-Person (FPs/PM)
· 1	5	341	18.5	18.4
2	5	271	17.2	15.8
3	4	375	12.5	30.0
4	5	356	18.6	19.1
5	4	311	10.0	31.2
6	5	414	16.0	25.9
7	4	311	15.2	20.4
8	3	143	15.6	9.2
9	6	195	19.9	9.8
10	5	91	10.7	8.5
			Average :	18.8
			Minimum :	8.5
			Maximum :	31.2

Figure 7.3.3 **Table - Productivity Rate**

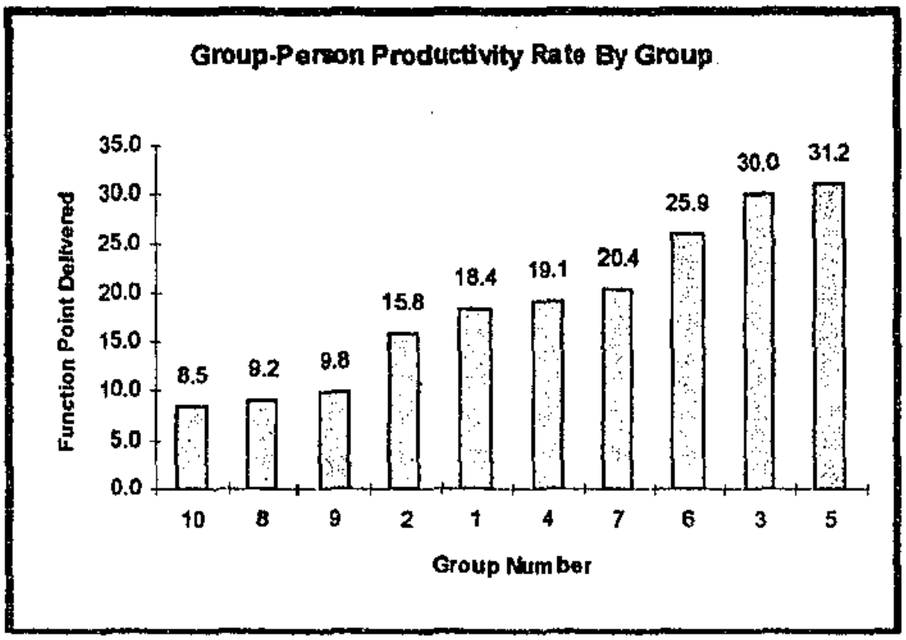


Figure 7.3.4 **Graph - Group-Person Productivity Rate By Group**

The data presented in this section (**Section 7.3**) will not be used in the remaining sections of this chapter. It will be used in **Chapter 10** for the final analysis.

7.4 STUDENT PROJECTS VS PROFESSIONAL PROJECTS

This section compares the project delivery rate (hours to deliver one function point) and productivity rate (function points per person-month) of student projects against projects developed by organisations from industry. The data presented in **Figures 7.4.1(a)** and **7.4.2(a)** are details of projects developed by organisations from the industry. It is taken from the Australian Software Metrics Association (ASMA) Project Database - Release 3 (1993b). In total, there are 86 projects from 15 organisations. The data below are taken from eight projects developed for the personal computer platform and are categorised in order to make a comparison of projects of a similar type. **Figures 7.4.1(b)** and **7.4.2(b)** further refined the data to present those projects that were developed using 4GL tools. The data presented in **Figures 7.4.3** and **7.4.4** are details of projects developed by the software engineering students. The data are presented in the format used by the Australian Software Metiics Association. This is to improve the means of comparing the results of students and professional projects. **Figure 7.4.5** presents a glossary of the terms used on the tables below.

ASMA ID	Delivery Rate (Hrs/FP)	Size (FP)	Hardware Platform	Time Recording		Development Type
				Level	Method	
4	18.5	502	PC	1	C	ND/X
9	1.7	917	PC	3	D	ND/X
10	1.9	273	PC	3	B	ND
11	1.3	220	PC	1	A	ND
16	5.5	1355	PC	1	E	CP/X
24	4.3	597	PC	1	C	ND/P1
49	2.3	1362	PC	3	B	ND
82	6.9	151	PC	1	A	ND/PS
Average :	5.3	672.1				
Minimum :	1.3	151				
Maximum :	18.5	1362				

Figure 7.4.1 (a) **Table - New Development Of Projects (ASMA, 1993b)**

ASMA ID	Delivery Rate (Hrs/FP)	Size (FP)	Hardware Platform	Time Recording		Development Type
				Level	Method	
9	1.7	917	PC	3	D	ND/X
10	1.9	273	PC	3	B	ND
11	1.3	220	PC	1	A	ND
24	4.3	597	PC	1	C	ND/P1
49	2.3	1362	PC	3	B	ND
Average :	**2.3**	**673.8**				
Minimum :	**1.3**	**220**				
Maximum :	**4.3**	**1362**				

Figure 7.4.1 (b) **Table - New Projects Developed Using 4GL Tools (ASMA, 1993b)**

ASMA ID	Year Implemented	DBMS	Language Type	Application Generator	CASE	Elapsed Time (Months)	Maximum Team Size
4	1991	Yes	3GL	No		14	8
9	1991	Yes	4GL	No	No	6	
10	1991	Yes	4GL	No	No	5	2
11	1991	Yes	4GL	No		3	2
16	1992	Yes	3GL	No	No		
24	1992	Yes	4GL	No		6	6
49	1992	Yes	4GL	No		11	4
82	1992	Yes	3GL	No	Yes	26	3

Figure 7.4.2 (a) **Table - Project Attributes (ASMA, 1993b)**

ASMA ID	Year Implemented	DBMS	Language Type	Application Generator	CASE	Elapsed Time (Months)	Maximum Team Size
9	1991	Yes	4GL	No	No	6	
10	1991	Yes	4GL	No	No	5	2
11	1991	Yes	4GL	No		3	2
24	1992	Yes	4GL	No		6	6
49	1992	Yes	4GL	No		11	4

Figure 7.4.2 (b) **Table - Project Attributes Of Projects Developed Using 4GL Tools (ASMA, 1993b)**

Group Number	Delivery Rate (Hrs/FP)	Size (FP)	Hardware Platform	Time Recording		Development Type
				Level	Method	
1	7	341	PC	1	B	ND
2	8.1	271	PC	1	B	ND
3	4.3	375	PC	1	B	ND
4	6.7	356	PC	1	B	ND
5	4.1	311	PC	1	B	ND
6	4.9	414	PC	1	B	ND
7	6.3	311	PC	1	B	ND
8	14	143	PC	1	B	ND
9	13.1	195	PC	1	B	ND
10	15.1	91	PC	1	B	ND
Average :	**8.4**	**280.8**				
Minimum :	**4.1**	**91**				
Maximum :	**15.1**	**414**				

Figure 7.4.3 **Table - New Development Projects (Student Projects)**

Group Number	Year Implemented	DBMS	Language Type	Application Generator	CASE	Elapsed Time (Months)	Maximum Team Size
1	1993	Yes	4GL	Yes	-	9	5
2	1993	Yes	4GL	Yes	-	9	5
3	1993	Yes	4GL	Yes	-	9	4
4	1993	Yes	4GL	Yes	-	9	5
5	1993	Yes	4GL	Yes	-	9	4
6	1993	Yes	4GL	Yes	-	9	5
7	1993	Yes	4GL	Yes	-	9	4
8	1993	Yes	4GL	Yes	-	9	3
9	1993	Yes	4GL	Yes	-	9	6
10	1993	Yes	4GL	Yes	-	9	5

Figure 7.4.4 **Table - Project Attributes (Student Projects)**

HARDWARE PLATFORM	DESCRIPTION
PC	Personal Computer
TIME RECORDING LEVEL	**DESCRIPTION**
1	Development Team
2	Level 1 plus Development Team Support
3	Level 2 plus Operating Support Centre
TIME RECORDING METHOD	**DESCRIPTION**
A	Staff Hours (Recorded)
B	Staff Hours (Derived)
C	"Productive" Time Only (Recorded)
D	A combination of methods
E	Neither A, B or C
DEVELOPMENT TYPE	**DESCRIPTION**
ND	New Development
ND/X	Unknown
ND/P1	New Development - Phase 1
ND/PS	New Development - Packaged Software
CP/X	Unknown

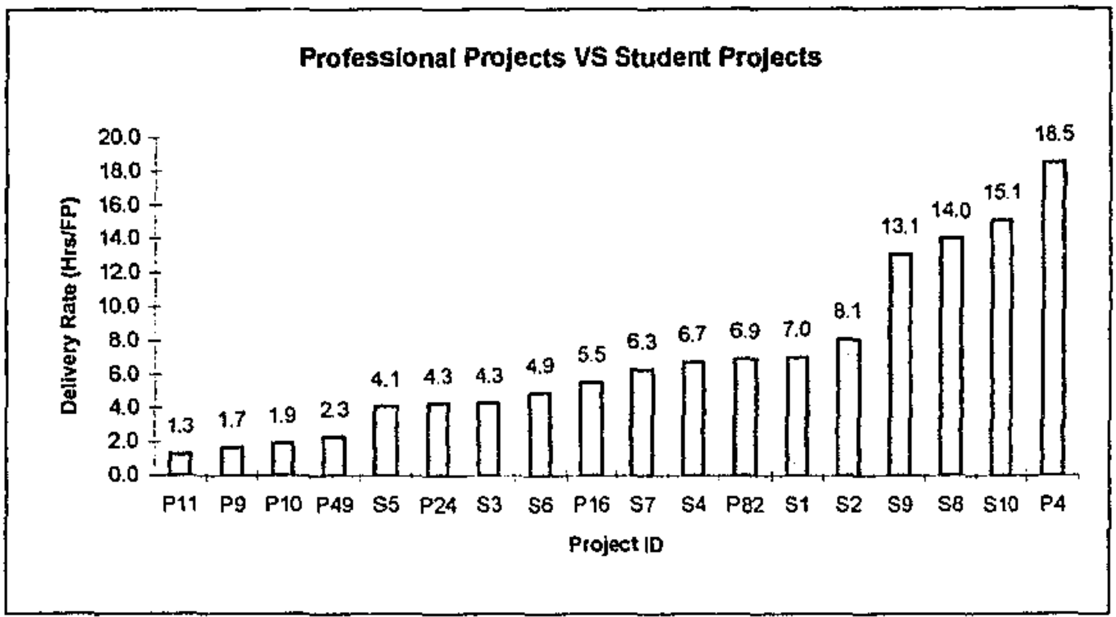
Figure 7.4.5 **Table - Glossary Of Terms Used By ASMA (1993b)**

Based on the results presented in the tables above, it is clear that professional developers are producing function points at a higher rate than students. The data from ASMA shows that professional developers took around 5.3 hours to deliver one function point, whereas the students took around 8.4 hours. Another interesting result showed that professional developers that used 4GL tools took around 2.3 hours to deliver one function point. The size of the software produced by the students are also relatively small when compared with the delivery rate and elapsed time. **Figures 7.4.6 (a)** and **7.4.7 (a)** combined the data from the ASMA and student projects. Similarly, **Figures 7.4.6 (b)** and **7.4.7 (b)** combined the data from ASMA projects developed using 4GL tools and student projects. The data in the table are sorted according to the projects' delivery rate. The Project ID with a prefix of "P" signifies a professional project and "S", a student project.

Project ID	Project Delivery Rate (Hrs/FP)	Size (FPs)	Elapsed Time (Months)	Maximum Team Size
P11	1.3	220	3	2
P9	1.7	917	6	-
P10	1.9	273	5	2
P49	2.3	1362	11	4
S5	4.1	311	9	4
P24	4.3	597	6	6
S3	4.3	375	9	4
S6	4.9	414	9	5
P16	5.5	1355	-	-
S7	6.3	311	9	4
S4	6.7	356	9	5
P82	6.9	151	26	3
S1	7.0	341	9	5
S2	8.1	271	9	5
S9	13.1	195	9	6
S8	14.0	143	9	3
S10	15.1	91	9	5
P4	18.5	502	14	8

Note : Prefix "P" signifies Professional Project and "S" signifies Student Project

Figure 7.4.6 (a) **Table - Professional Projects VS Student Projects**



Note : Prefix "P" signifies Professional Project and "S" signifies Student Project

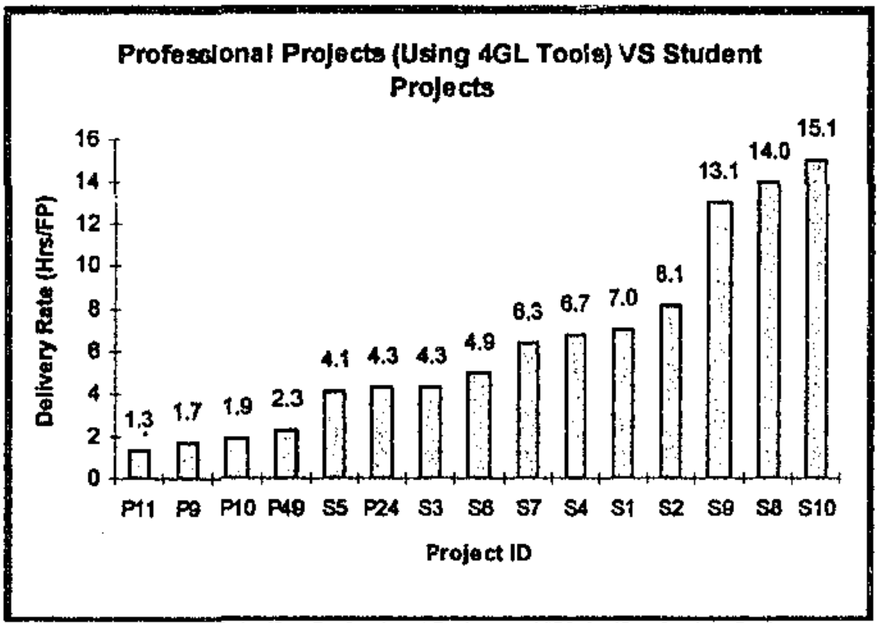
Figure 7.4.7 (a) **Graph - Delivery Rate Of Professional & Student Projects**

Based on the table above, almost 75 per cent of the professional projects are on the upper half of the table and the majority of the students' projects are on the lower half. As mentioned before, it clearly shows that professional developers are more productive. The two professional projects (P82 and P4) appear to be less productive. However, it is important to note that P82 took 26 months to deliver 151 function points, and P4 took 14 months to deliver 502 function points. Furthermore, both projects were developed using 3GL languages without the aid of an application generator.

Project ID	Project Delivery Rate (Hrs/FP)	Size (FPs)	Elapsed Time (Months)	Maximum Team Size
P11	1.3	220	3	2
P9	1.7	917	6	-
P10	1.9	273	5	2
P49	2.3	1362	11	4
S5	4.1	311	9	4
P24	4.3	597	6	6
S3	4.3	375	9	4
S7	6.3	311	9	4
S1	7.0	341	9	5
S6	4.9	414	9	5
S8	14.0	143	9	3
S10	15.1	91	9	5
S2	8.1	271	9	5
S4	6.7	356	9	5
S9	13.1	195	9	6

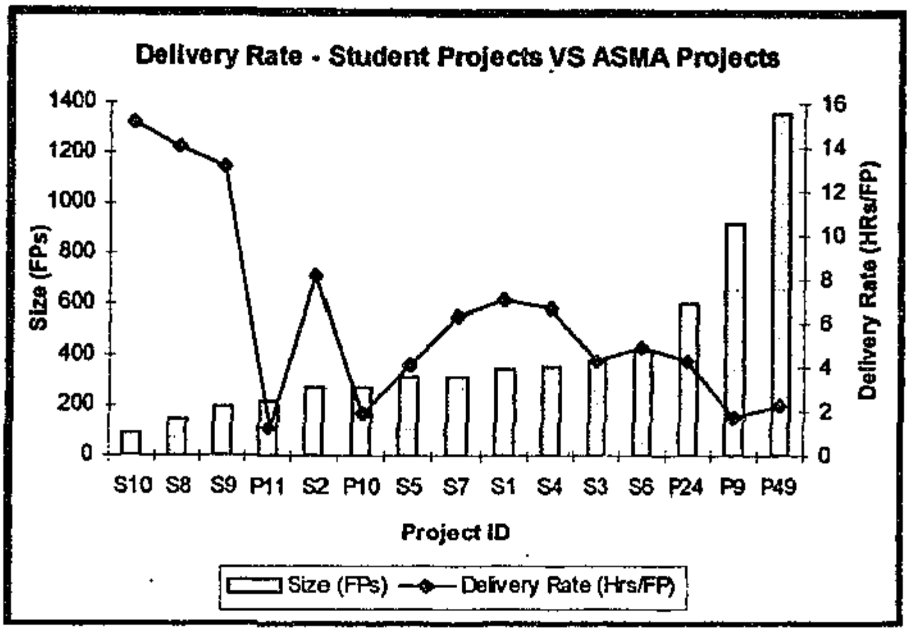
Note : Prefix "P" signifies Professional Project and "S" signifies Student Project

Figure 7.4.6 (b) **Table - Professional Projects (Using 4GL Tools) VS Student Projects**



Note : Prefix "P" signifies Professional Project and "S" signifies Student Project

Figure 7.4.7 (b) **Graph - Delivery Rate Of Professional (Using 4GL Tools) & Student Projects**



Note : Prefix "P" signifies Professional Project and "S" signifies Student Project

Figure 7.4.8 **Graph - Delivery Rate VS Size**

Figures 7.4.6 (b) and **7.4.7 (b)** show all professional projects developed using 4GL tools are on the upper half of the table. **Figure 7.4.8** provide further supporting evidences that professional developers are more productive.

The table below (**Figure 7.4.9**) is taken from Caper Jones (1991, p. 454). The data was collected by Caper Jones' company Software Productivity Research (SPR). The main objective for having this data is to enable a comparison between the productivity rate of professional projects against the student projects. As before, the data from the students' projects are collected and presented in the format used by the SPR. However, to ensure a reasonable comparison, the number of person-months was redefined. The number of hours each student could spent was set to 40, bringing a total of 160 hours per month. The data from the students' projects are presented in **Figure 7.4.10.**

Code	Technology	Type	Size (FPs)	Effort (Person-Months)	Schedule (Elapsed Months)	Staff	Documentation	Productivity (FPs/PM)
A	MF	D	588	40.3	10.5	3.8	1959	14.58
B	MF	D	193	13.8	6.0	2.3	199	13.99
C	MF	D	145	16.0	3.5	4.6	166	9.06
D	MF	E	63	5.2	2.0	2.6	163	12.12
E	MF	D	69	16.2	3.7	4.4	174	4.26
F	MF	D	437	110.0	8.0	14.4	335	3.80
G	MF	M	288	30.8	5.3	5.9	380	9.36
H	MF	E	604	164.3	22.0	6.6	1000	3.68
J	PC	D	392	34.0	11.0	3.1	914	11.53
K	MF	E	202	12.2	2.5	3.2	67	16.50
L	MF	E	57	6.2	5.0	1.5	108	9.83
M	PC	E	80	50.0	11.3	6.3	1407	1.60
N	MF	D	79	7.1	7.0	1.0	121	11.06
P	PK	E	513	104.4	7.0	14.9	1181	4.91
Q	MF	E	671	186.9	16.0	11.7	1541	3.59
R	MF	M	3162	120.0	12.0	10.1	2136	25.98
S	MF	D	158	28.5	5.7	4.8	450	5.54
T	MF	D	63	14.2	4.3	3.3	110	4.44
U	PC	E	405	35.0	5.3	7.0	1195	10.95
	Average :		429	52.4	7.8	6.0	716	9.30

KEY :

TYPE	TECHNOLOGY
D = Development	MF = Mainframe
E = Enhancement	PC = Micro Computer
M = Maintenance	PK = Package

Figure 7.4.9 **Table - Productivity Data Taken From SPR (Jones, 1991)**

Code	Technology	Type	Size (FPs)	Effort (Person-Months)	Schedule (Elapsed Months)	Staff	Documentation	Productivity (FPs/PM)
1	PC	D	341	14.8	9.0	5.0	-	23.02
2	PC	D	271	13.8	9.0	5.0	-	19.71
3	PC	D	375	10.0	9.0	4.0	-	37.50
4	PC	D	356	14.9	9.0	5.0	-	23.93
5	PC	D	311	8.0	9.0	4.0	-	38.97
6	PC	D	414	12.8	9.0	5.0	-	32.36
7	PC	D	311	12.2	9.0	4.0	-	25.52
8	PC	D	143	12.5	9.0	3.0	-	11.44
9	PC	D	195	15.9	9.0	6.0	-	12.24
10	PC	D	91	8.6	9.0	5.0	-	10.63
	Average :		281	12.3	9.0	4.6	-	23.53

KEY :

TYPE	TECHNOLOGY
D = Development	PC = Micro Computer

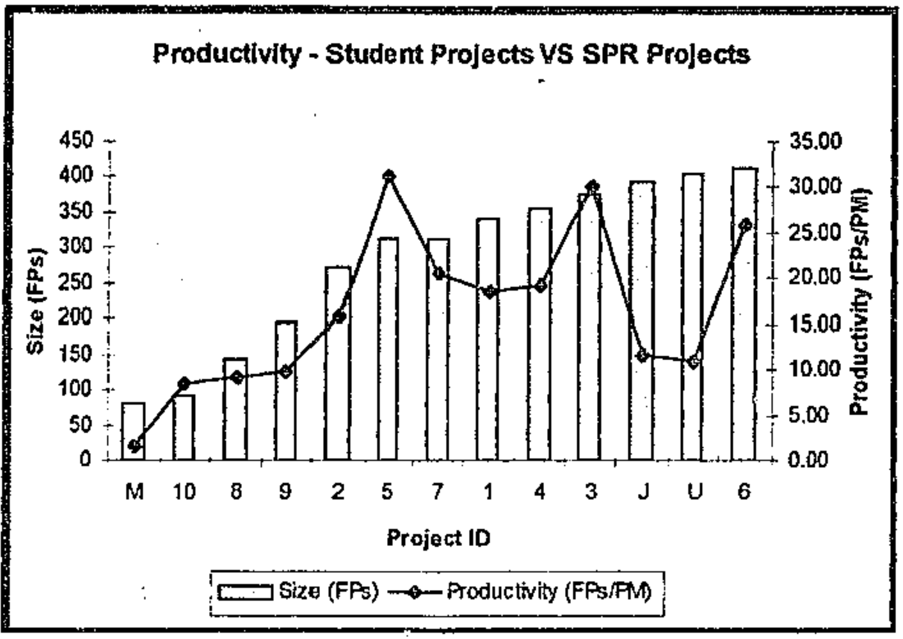
Figure 7.4.10 **Table - Productivity Data Of Student Projects**

The data from SPR is different from that obtained from the ASMA. Here, the students appear to be more productive than those projects from the SPR. The students were delivering around 23 function points per person-month, whereas the SPR projects were only delivering around 9 function points. It is important to point out that the students' projects were not completed. Out of the 19 projects from SPR, 3 were developed for the personal computer platform. Among the enhancements. When this data is separated from the main table (See **Figure 7.4.11**), a separate set of averages were derived. Again, it shows that the SPR projects were delivering less function points per person-month (around 8 function points per person-month). The average software size, elapsed month and number of staff were very similar. But the effort put in by the SPR projects were very high (around 40 person-months) when compared to the students' projects (around 15 person-months). Since many details about the projects from SPR were kept confidential (Jones, 1991), it is not possible to determine what causes the low delivery rate. One possible reason could be the different method that SPR used for deriving the project function points.

Code	Technology	Type	Size (FPs)	Effort (Person-Months)	Schedule (Elapsed Months)	Staff	Documen-tation	Productivity (FPs/PM)
J	PC	D	392	34.0	11.0	3.1	914	11.53
M	PC	E	80	50.0	11.3	6.3	1407	1.60
U	PC	E	405	35.0	5.3	7.0	1195	10.95
	Average :		292	39.7	9.2	5.5	1172	8.03

Figure 7.4.11 **Table - Productivity Data From SPR PC Projects (Jones, 1991)**

The graph in **Figure 7.4.12** presents the productivity of both sets of projects together with the size of the software. As mentioned before, SPR projects were producing rather large software but their productivity rate was quite low. Though the majority of the students' productivity rate were quite reasonable, there were a few that were very low and had a small software size.



Note : Project ID in alphabet refers to SPR projects, those in numerical value refer to student projects.

Figure 7.4.12 **Graph - Productivity Rate : Student Projects VS SPR Projects**

7.5 <u>SUMMARY</u>

Figure 7.5.1 presents the compilation of productivity rates derived from the sections above. Groups with high productivity rates are represented in **bold** typeface and those with low productivity rates are represented in ***bold-italic*** typeface.

Group Number	Productivity Rate Of Each Group (FPs/PM)	Productivity Rate Of Each Student (FPs/PM)	Project Delivery Rate (Hrs/FP)
1	23.0	4.6	7.0
2	19.7	3.9	8.1
3	**37.5**	**9.4**	**4.3**
4	23.9	4.8	6.7
5	**39.0**	**9.7**	**4.1**
6	**32.4**	**6.5**	**4.9**
7	25.5	6.4	6.3
8	***11.4***	***3.8***	***14.0***
9	***12.2***	***2.0***	***13.1***
10	***10.6***	***2.1***	***15.1***

Figure 7.5.1 **Table - Overall Productivity Rate**

Groups 8 and 10 had the lowest delivery rate, each taking 14 and 15.1 hours to deliver one function point, respectively. Groups 3 and 5 had the highest delivery rate. They were delivering around 30 function points per person-month, with each student producing around 7 function points. The size of the their software was well above the average size of 280.8 function points.

When the statistics of students' projects were compared with the statistics of professional projects from the ASMA, it shows that the students took a longer time to deliver one function point, at a rate lower than the ASMA average. Yet when the same statistics were compared with statistics of projects from SPR, the students were delivering higher function points per person-month. Of course, when comparing statistics like these, there are other aspects which need to be taken into consideration, aspects such as the type of applications being developed and the type of software development platform used. This information was not known for the SPR projects.

CHAPTER 8 : MEASURING SOFTWARE QUALITY

8.1 MEASURING SOFTWARE QUALITY

It is difficult, if not impossible, to develop software that is totally perfect. There will always be some problems. Some of these problems can be easily fixed whilst others may require a considerable amount of rework. The software that was produced by the students was certainly no different. This chapter will present the quality of the software based on the number of defects found. The equation used to derive the quality of these software are taken from Pressman (1992, p. 47).

$$\text{Quality} = \frac{\text{Defects}}{\text{Function Points}}$$

The definition of defects may vary from person to person. In this case the defects have been classified as follows :

- ❑ Any operation that causes the application to "halt" (in Microsoft Access) or terminate during processing without making any changes to the external logical files.
- ❑ Any operation that was reported to be successful but failed to complete or achieve its designated task(s).
- ❑ Any defects that were detected during the evaluation process (see **Appendix A** for more information).

Functions that were presented in the software menus but not implemented were not counted as defects because they were not counted as function points. Defects were only counted on functions that were delivered.

8.2 QUALITY OF THE SOFTWARE

This section presents the quality of the software evaluated. For more information on the types of defects or bugs that were found, see **Appendix A**. The results on the quality of the software are presented in **Figures 8.2.1** and **8.2.2**.

Group Number	Size (FPs)	Defects (D)	Quality (D/FPs)
1	341	3	0.0088
2	271	21	0.0775
3	375	10	0.0267
4	356	6	0.0169
5	311	2	0.0064
6	414	16	0.0386
7	311	4	0.0129
8	143	2	0.0140
9	195	4	0.0205
10	91	9	0.0989
	Average :	7.7	0.0321
	Minimum :	2	0.0064
	Maximum :	21	0.0989

Figure 8.2.1 **Table - Determining Software Quality**

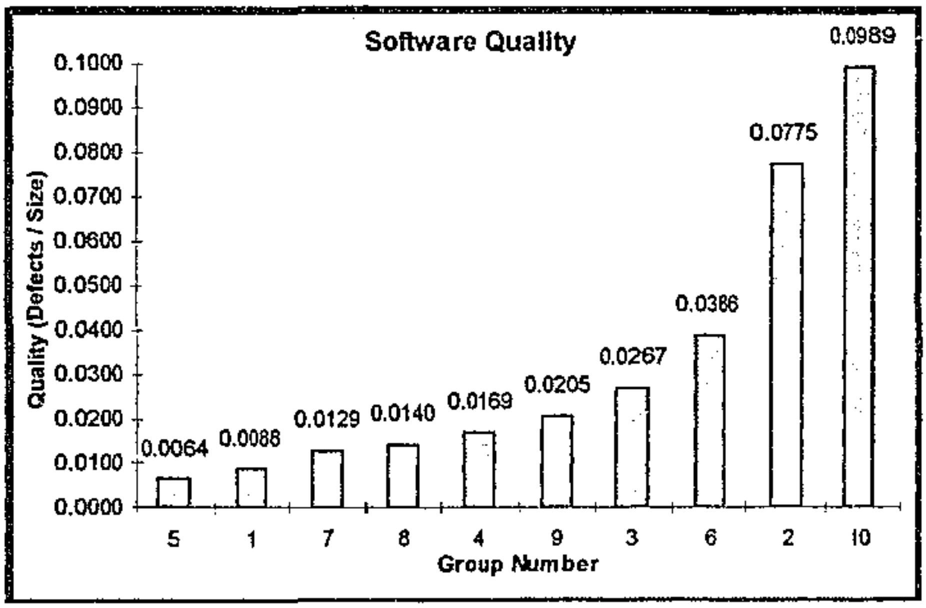


Figure 8.2.2 **Graph - Software Quality**

9.1 MEASURING SOFTWARE USABILITY

This chapter describes the method used for determining the usability and learnability of the software produced by each project group. It will also present the results derived from the usability exercise.

9.1.1 Usability Exercise

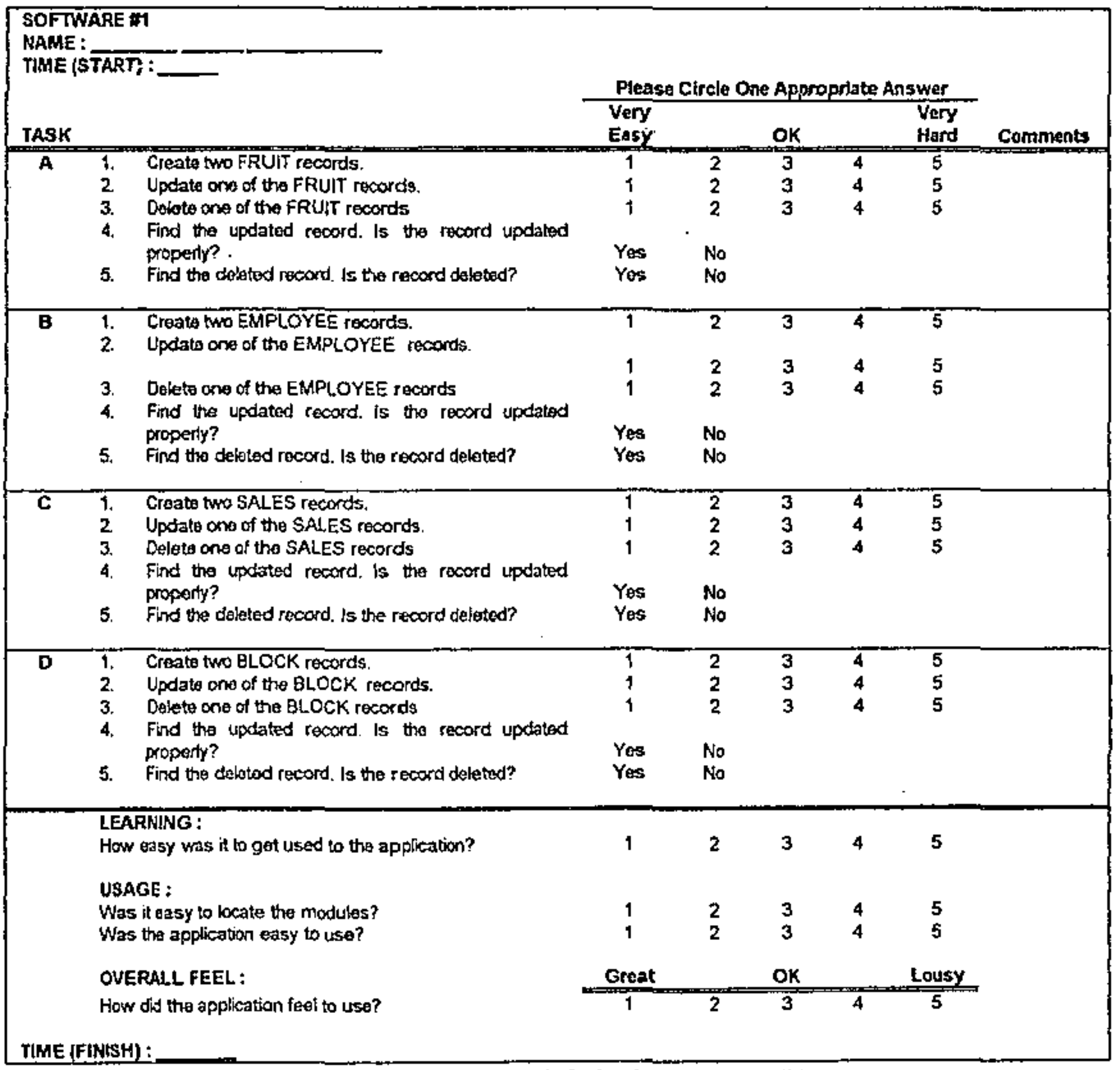
The usability exercise was conducted by presenting a group of five independent students with a list of tasks to be performed by each application. The students that took part in this exercise were required to have some background with Microsoft Windows. Students in the 1993 Software Engineering Project were not allowed to participate. Each student was given 30 minutes to perform the tasks specified.

Since each of the applications covers different aspects of the orchard project, it was difficult to create a generic usability test plan. Therefore for each application, a unique set of tasks was provided. This set of tasks consists of four main sections. Each section focused on one module of the application. The tasks to be performed included creating, deleting and updating of records. The students were also required to check whether an updated record was indeed updated, and a deleted record was deleted.

The students were required to log their start and finish time for each set of tests. At the end of the test, the students were asked to comment to their perception of the application's usability and learnability.

9.1.2 Usability Test Plan

This section describes the format of the usability test plan and its format on how data were collected from each student (See **Figure 9.1.2.1**). The tasks specified from A to D vary from application to application, although the objective of its operations remain the same.



SOFTWARE #1
NAME : ____________
TIME (START) : _____

			Please Circle One Appropriate Answer					
			Very				Very	
TASK			Easy		OK		Hard	Comments
A	1.	Create two FRUIT records.	1	2	3	4	5	
	2.	Update one of the FRUIT records.	1	2	3	4	5	
	3.	Delete one of the FRUIT records	1	2	3	4	5	
	4.	Find the updated record. Is the record updated properly? .	Yes	No				
	5.	Find the deleted record. Is the record deleted?	Yes	No				
B	1.	Create two EMPLOYEE records.	1	2	3	4	5	
	2.	Update one of the EMPLOYEE records.	1	2	3	4	5	
	3.	Delete one of the EMPLOYEE records	1	2	3	4	5	
	4.	Find the updated record. Is the record updated properly?	Yes	No				
	5.	Find the deleted record. Is the record deleted?	Yes	No				
C	1.	Create two SALES records.	1	2	3	4	5	
	2.	Update one of the SALES records.	1	2	3	4	5	
	3.	Delete one of the SALES records	1	2	3	4	5	
	4.	Find the updated record. Is the record updated properly?	Yes	No				
	5.	Find the deleted record. Is the record deleted?	Yes	No				
D	1.	Create two BLOCK records.	1	2	3	4	5	
	2.	Update one of the BLOCK records.	1	2	3	4	5	
	3.	Delete one of the BLOCK records	1	2	3	4	5	
	4.	Find the updated record. Is the record updated properly?	Yes	No				
	5.	Find the deleted record. Is the record deleted?	Yes	No				
		LEARNING :						
		How easy was it to get used to the application?	1	2	3	4	5	
		USAGE :						
		Was it easy to locate the modules?	1	2	3	4	5	
		Was the application easy to use?	1	2	3	4	5	
		OVERALL FEEL :	Great		OK		Lousy	
		How did the application feel to use?	1	2	3	4	5	

TIME (FINISH) : _____

Figure 9.1.2.1 **Table - Format Of Usability Test Plan**

9.1.3 Deriving Usability Of The Application

Not all the students were able to perform all the tasks specified. The results from each task varied from student to student. Some were able to perform a task successfully while others encountered problems. Therefore to gather the score for each task, an average was derived. Averages were also derived for the applications' LEARNABILITY, USAGE and OVERALL FEEL. Responses with "YES" or "NO" were calculated by deriving a percentage based on the number of "YES" responses. This is to determine the percentage of the application's operation success rate. The amount of time taken was also calculated in terms of minutes. **Figure 9.1.3.1** presents the raw data collected from the exercise.

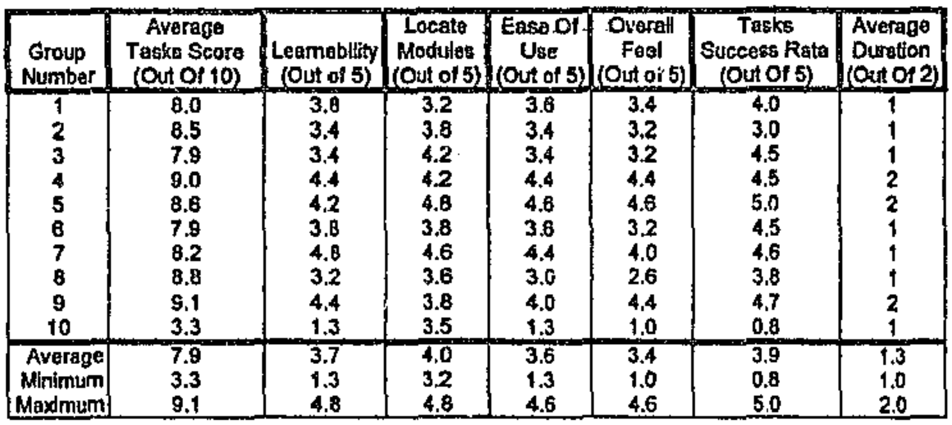
Group Number	Average Tasks Score (Out Of 60)	Learnability (Out of 5)	Locate Modules (Out of 5)	Ease Of Use (Out of 5)	Overall Feel (Out of 5)	Tasks Success Rate (Out Of 800%)	Average Duration (Minutes)
1	47.8	3.8	3.2	3.6	3.4	640	32.8
2	50.8	3.4	3.8	3.4	3.2	480	29.8
3	47.2	3.4	4.2	3.4	3.2	720	32.2
4	54.0	4.4	4.2	4.4	4.4	720	19.6
5	51.8	4.2	4.8	4.6	4.6	800	19.2
6	47.7	3.8	3.8	3.6	3.2	715	31.4
7	49.0	4.8	4.6	4.4	4.0	740	21.8
8	52.8	3.2	3.6	3.0	2.6	600	22.4
9	54.6	4.4	3.8	4.0	4.4	747	20.2
10	19.8	1,3	3.5	1.3	1.0	125	22.2
Average	47.5	3.7	4.0	3.6	3.4	629	25.2
Minimum	19.8	1.3	3.2	1.3	1.0	125	19.2
Maximum	54.6	4.8	4.8	4.6	4.6	800	32.8

Figure 9.1.3.1 **Table - Raw Usability Data**

After the raw data were derived, some of the data were scaled down to a more reasonable range, and the data were weighted as follows :

• Average tasks score	10
• Learnability	5
• Locate modules	5
• Ease of use	5
• Overall feel	5
• Task success rate	5
• Average duration	2

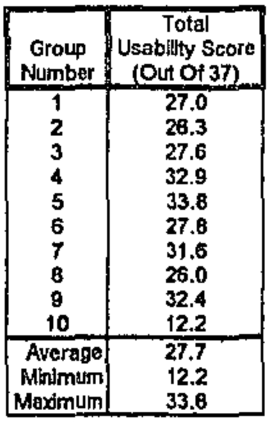
For the Average Duration, time ranges between 11 to 20 minutes are scored as 2 and 21 to 30+ minutes as 1. The weightings were determined by a subjective assessment of the importance if each element. **Figure 9.1.3.2** presents the final set of data derived.



Group Number	Average Tasks Score (Out Of 10)	Learnability (Out of 5)	Locate Modules (Out of 5)	Ease Of Use (Out of 5)	Overall Feel (Out of 5)	Tasks Success Rate (Out Of 5)	Average Duration (Out Of 2)
1	8.0	3.8	3.2	3.6	3.4	4.0	1
2	8.5	3.4	3.8	3.4	3.2	3.0	1
3	7.9	3.4	4.2	3.4	3.2	4.5	1
4	9.0	4.4	4.2	4.4	4.4	4.5	2
5	8.6	4.2	4.8	4.6	4.6	5.0	2
6	7.9	3.8	3.8	3.6	3.2	4.5	1
7	8.2	4.8	4.6	4.4	4.0	4.6	1
8	8.8	3.2	3.6	3.0	2.6	3.8	1
9	9.1	4.4	3.8	4.0	4.4	4.7	2
10	3.3	1.3	3.5	1.3	1.0	0.8	1
Average	7.9	3.7	4.0	3.6	3.4	3.9	1.3
Minimum	3.3	1.3	3.2	1.3	1.0	0.8	1.0
Maximum	9.1	4.8	4.8	4.6	4.6	5.0	2.0

Figure 9.1.3.2 **Table - Adjusted Usability Data**

After all the necessary data were adjusted, the total usability score for each application was derived. These scores were derived by calculating the sum of all the data presented in **Figure 9.1.3.2**, of each application. The total usability scores were presented out of 37. **Figure 9.1.3.3** and **9.1.3.4** presents the total usability score of each application.



Group Number	Total Usability Score (Out Of 37)
1	27.0
2	26.3
3	27.6
4	32.9
5	33.8
6	27.8
7	31.6
8	26.0
9	32.4
10	12.2
Average	27.7
Minimum	12.2
Maximum	33.8

Figure 9.1.3.3 **Table - Total Usability Score**

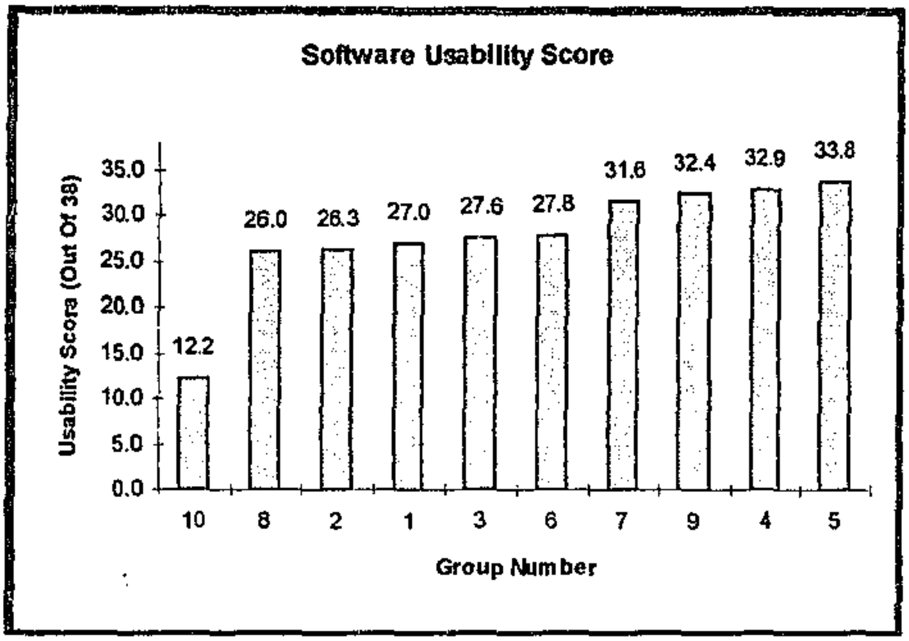


Figure 9.1.3.4 **Graph - Total Usability Score**

Based on the average score, the majority of the applications did very well from the usability test, with the exception of **Group 10**'s application. It scored the lowest and its well below the average score.

CHAPTER 10 : FINAL ANALYSIS AND CONCLUSION

10.1 FINAL ANALYSIS

All the data necessary for the final analysis has been collected and presented in chapters 3 to 9. Information on software size and the number of defects found has been derived. Albrecht's function point method has been used to measure the size of the software. At the same time, guidelines have been established to ensure that the software was being measured consistently. As for the number of defects, the same approach was derived from the tests performed on each piece of software. The usability of the software was determined by asking a group of students to perform a series of tasks on each piece of software and provide feedback on its usability, through the use of a questionnaire. Members of the judging panel provided the information on team scores, solution functionality and solution quality. The students provided the information on cross scores. The students' course averages were obtained from the university's Student Services Department. The remaining information was gathered from the software engineering students by means of questionnaires. It has to be stated that the accuracy of this information is dependant upon the accuracy of the data provided by the students. The information gathered has been divided into 30 sub-categories which fall under two major categories, Process[16] and Product[17]. These sub-categories are listed below :

PROCESS

- Team size
- Total hours spent
- Requirement time
- Analysis time
- Design time
- Coding time
- Testing time
- Requirement time (%)
- Analysis time (%)
- Design time (%)
- Coding time (%)
- Testing time (%)
- Project management
- On schedule
- Team work
- Contribution to project
- Cross scores
- Average age
- Course averages
- Female students (%)
- Part-time students (%)
- Staff adviser's advice
- APT methodology
- Development software
- Access to client
- Productivity Rate

[16] Process is a metric of numerical value that describes a software process such as the amount of time required to code a piece of software. **(For more information, refer to Section 2.2.2)**

[17] Product is a metric of numerical value that is extracted or derived from a piece of software. **(For more information, refer to Section 2.2.2)**

PRODUCT

- Solution functionality
- Software usability
- Solution quality
- Defects found
- Software size

10.1.1 Description Of Information

Process

□ Team size
The size of each group varied from 3 to 6 students.

□ Total hours spent
The average total hours spent on the project by each group.

□ Requirement time
The total hours spent on the requirement phase by each group. **(See Section 2.1.2.1 for the explanation of the development phases)**

□ Analysis time
The total hours spent on the analysis phase by each group.

□ Design time
The total hours spent on the design phase by each group.

□ Coding time
The total hours spent on the coding phase by each group.

□ Testing time
The total hours spent on the testing phase by each group.

□ Requirement time (%)
The percentage of time spent on the requirement phase by each group.

□ Analysis time (%)
The percentage of time spent on the analysis phase by each group.

□ Design time (%)
The percentage of time spent on the design phase by each group.

□ Coding time (%)
The percentage of time spent on the coding phase by each group.

□ Testing time (%)
The percentage of time spent on the testing phase by each group.

❑ Project management
The average score on how satisfied each group was with the management of their project. The score was out of 10.

❑ On schedule
Whether each group felt they were able to complete their project on schedule.

❑ Team work
The average score on how satisfied each group was with the way the group operated. The score was out of 10.

❑ Contribution to project
The average score on how satisfied each team member was with their contribution being received by the rest of the team. The score was out of 10.

❑ Cross scores
Part of the assessment of the project involved each team member giving a score (out of 28) for the contribution made by other team members. The cross score is the average of all the scores given by the members of each group.

❑ Average age
The average age of the students in each group.

❑ Courses averages
The average of all the course averages of students in each group.

❑ Female students (%)
The percentage of female students in each group.

❑ Part-time students (%)
The percentage of part-time students in each group.

❑ Staff adviser's advice
The average score on how satisfied each group was with their staff adviser's advice. The score was out of 10.

❑ APT methodology
The average score on how satisfied each group was with the use of the APT methodology. The score was out of 10.

□ Development software
Each group had to select their own software platform. This is the average score on how satisfied each group was with the development software chosen. The score was out of 10.

□ Access to client
The average score on how satisfied each group was with the method(s) used for communicating with the client. The score was out of 10.

□ Productivity Rate
The delivery rate of function points per person-month. **(For more information, refer to Chapter 7)**

Product

□ Solution functionality
The average score of each piece of software's functionality which was awarded by the judging panel. The score was out of 25.

□ Solution quality
The average score of each piece of software's quality which was awarded by the judging panel. The score was out of 25.

□ Software size
The size of each piece of software (in function points) was derived by the investigator using Albrecht's Function Point Analysis. **(For more information, refer to Chapter 6)**

□ Software usability
The average score of each piece of software's usability which was derived after conducting a series of usability tests. The score was out of 37. **(For more information, refer to Chapter 9)**

□ Defects found
The number of defects found on each piece of software was derived after conducting a series of tests. **(For more information, refer to Chapter 8)**

10.2 STATISTICAL METHOD USED

The next phase, after all the information had been compiled, was to apply some statistical measurement of correlation to this information. The aim of this was to determine the relationship (if any), between the information categorised above. For example, based on this study, questions which could be raised, are, "Does more time spent by students on software testing affect the quality of the software?" or "Would more time spent by students on coding produce software of a bigger size?". These are some of the questions that will be addressed.

In order to obtain the answers to these and other questions, four types of statistical methods were considered. These methods were : linear regression, Pearson's correlation coefficient, Spearman's rank correlation coefficient and Kendall's rank-order correlation coefficient. Out of these four statistical methods, Spearman's rank correlation coefficient method was selected for the research. The results obtained through the use of the linear regression approach and Pearson's correlation coefficient method were abnormally influenced by the outliers[18] in the data. These two methods are more suitable for normally-distributed attribute values. Some useful results were obtained using Kendall's rank-order correlation coefficient method. However, these were not sufficient for the purposes of arriving at any major conclusions.

The Spearman's rank correlation coefficient method was chosen because it produced sufficient results that were able to address the questions proposed by the research. The Spearman's rank correlation coefficient is very similar to the Pearson's correlation coefficient method, except that the former is a *robust measure*. The use of a robust measure is preferred because "... most software measurements are not normally-distributed and usually contain atypical values ..." (Fenton, 1991, p. 102). The rank correlation coefficient method is not easily influenced by both abnormal values and non-linearity of the underlying

[18] Outliers' data are those that are abnormally high or low in a series of data.

relationship. It is also not inclined to be influenced by very large values. The main difference between Spearman's and Pearson's method is that, the former calculates the correlation coefficient based on the rank of the attribute values whereas the latter is based on the raw values. Spearman's is considered better for "behavioural" data, which best describes the data used here, where large sections were obtained from survey material.

Spearman's rank correlation coefficient is denoted by r_s. This can be derived at by using the formula presented below (Freund et al., 1992, p. 511).

$$r_s = 1 - \frac{6(\sum d^2)}{n(n^2 - 1)}$$

The rank correlation coefficient for a given set of n pairs of x's and y's is calculated within several steps, where x and y are the attribute values (Freund et al., 1992, p. 511). First the x's and y's are ranked among themselves from low to high (or high to low). In this exercise, the values were rank in ascending order. The rank was obtained by giving the smallest attribute value the rank value of 1, the next rank value of 2 and so on. In the event where two or more attribute values are the same, an average of the related rank values is derived and assigned to these attribute values. The value for d is derived from the differences between the ranks and is substituted into the formula (Freund et al., 1992, p. 511). The correlation coefficient value varies from -1 to 1, where 1 indicates a perfect positive linear relationship, -1 indicates a perfect negative linear relationship, and 0 indicates no relationship (Fenton, 1991, p. 102). The results from the analysis are presented in **Figure 10.2.1**.

		PRO CESS																										PRO DUCT				
		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
PROCESS	Team Size 1																															
	Total Hours Spent 2	0.88																														
	Requirement Time 3	0.14	0.60																													
	Analysis Time 4	0.47	0.82	0.60																												
	Design Time 5	0.09	0.70	0.64	0.83																											
	Coding Time 6	-0.28	-0.13	-0.09	-0.55	-0.33																										
	Testing Time 7	0.63	0.96	0.48	0.70	0.55	0.07																									
	Requirement Time (%) 8	0.05	0.03	0.65	0.33	0.20	-0.48	-0.15																								
	Analysis Time (%) 9	0.30	0.10	0.27	0.56	0.20	-0.78	0.01	0.65																							
	Design Time (%) 10	-0.01	0.58	0.62	0.79	0.95	-0.39	0.42	0.25	0.28																						
	Coding Time (%) 11	-0.56	-0.81	-0.59	-0.98	-0.77	0.66	-0.66	-0.35	-0.58	-0.73																					
	Testing Time (%) 12	0.66	0.73	0.09	0.41	0.19	0.19	0.85	-0.50	-0.16	0.12	-0.41																				
	Project Management 13	0.19	0.28	0.01	0.56	0.50	-0.64	0.14	-0.08	0.36	0.55	-0.80	0.17																			
	On Schedule 14	-0.45	-0.19	0.11	-0.11	0.27	0.04	-0.27	-0.04	-0.34	0.42	0.11	-0.27	0.27																		
	Team Work 15	0.51	0.84	0.18	0.63	0.05	-0.41	0.68	-0.18	0.21	0.55	-0.69	0.65	0.79	0.23																	
	Contribution To Project 16	0.34	0.45	-0.01	0.57	0.58	-0.54	0.34	-0.23	0.15	0.60	-0.63	0.38	0.92	0.34	0.87																
	Cross Scores 17	0.48	0.72	0.25	0.49	0.44	0.19	0.65	-0.25	-0.02	0.31	-0.42	0.68	-0.10	-0.34	0.37	0.12															
	Average Age 18	-0.49	-0.36	0.02	-0.49	-0.06	0.47	-0.36	0.14	-0.42	-0.09	0.55	-0.58	-0.62	0.27	-0.62	-0.53	-0.06														
	Course Average 19	-0.15	0.16	-0.12	0.32	0.49	-0.27	0.18	-0.19	0.10	0.37	-0.28	-0.04	0.29	-0.04	0.33	0.32	0.42	0.00													
	Female Students (%) 20	0.06	0.12	0.03	-0.12	-0.31	0.50	0.21	-0.25	-0.25	-0.44	0.19	0.35	-0.22	-0.52	-0.22	-0.35	0.02	-0.27	-0.32												
	Part-Time Students (%) 21	-0.04	-0.05	0.20	-0.16	0.10	0.01	-0.25	0.36	-0.24	0.03	0.11	-0.44	-0.09	0.30	-0.25	-0.03	-0.38	0.56	-0.29	-0.17											
	Staff Adviser's Advice 22	0.78	0.55	0.12	0.61	0.22	-0.53	0.49	0.05	0.50	0.16	-0.67	0.50	0.87	-0.34	0.72	0.62	0.20	-0.77	0.02	0.15	-0.14										
	APT Methodology 23	-0.26	-0.02	0.17	0.10	0.22	0.04	-0.07	-0.12	-0.12	0.32	-0.04	-0.01	0.69	0.61	0.46	0.45	-0.35	-0.23	-0.12	0.11	0.06	0.21									
	Development Software 24	-0.21	0.25	0.37	0.47	0.61	-0.10	0.23	-0.08	0.09	0.71	-0.37	0.14	0.64	0.59	0.65	0.57	0.09	-0.25	0.24	-0.16	-0.22	0.19	0.81								
	Access To Client 25	-0.09	0.43	0.49	0.46	0.42	-0.12	0.37	0.05	-0.02	0.40	-0.46	0.29	0.19	0.12	0.26	0.18	0.10	-0.44	0.17	0.24	-0.30	0.05	0.20	0.35							
	Productivity Rate 26	-0.22	-0.31	-0.42	-0.58	-0.42	0.62	-0.05	-0.65	-0.63	-0.46	0.71	0.25	-0.32	0.21	-0.14	-0.32	0.15	0.25	0.03	0.48	-0.17	-0.33	0.22	0.14	-0.20						
PRODUCT	Solution Functionality 27	-0.13	0.21	0.04	0.09	0.27	0.52	0.39	-0.45	-0.25	0.29	0.07	0.38	-0.16	0.06	0.09	-0.03	0.60	0.23	0.23	-0.02	-0.41	-0.27	0.00	0.36	-0.11	0.67					
	Solution Quality 28	0.35	0.60	-0.02	0.38	0.47	0.17	0.71	-0.58	-0.72	0.42	-0.32	0.71	0.26	0.08	0.62	0.61	0.61	-0.13	0.42	-0.17	-0.34	0.22	0.04	0.40	-0.01	0.28	0.74				
	Software Size 29	-0.01	0.12	-0.07	-0.32	-0.21	0.80	0.30	-0.56	-0.66	-0.32	0.43	0.43	-0.38	-0.15	-0.15	-0.30	0.33	0.22	-0.22	0.66	-0.03	-0.14	0.15	0.00	-0.15	0.79	0.65	0.34			
	Software Usability 30	-0.02	0.03	-0.39	-0.08	0.03	0.32	0.26	-0.65	-0.24	-0.03	0.16	0.32	0.02	-0.04	0.32	0.12	0.69	0.03	0.63	-0.08	-0.55	-0.03	0.00	0.25	-0.21	0.60	0.68	0.72	0.39		
	Defects Found 31	0.41	0.12	0.09	-0.06	-0.45	0.16	0.16	0.19	0.15	-0.59	0.05	0.20	-0.38	-0.80	-0.31	-0.50	0.03	-0.24	-0.40	0.70	-0.04	0.31	-0.31	-0.56	0.00	0.04	-0.33	-0.39	0.31	-0.28	

Legends : ▨ Positive Correlation Coefficient ▨ Negative Correlation Coefficient

Figure 10.2.1 Results Obtained Using Spearman's Rank Correlation Coefficient

10.3 CONCLUSIONS DERIVED FROM THE ANALYSIS

The conclusions derived from the analysis are presented in themes. In this study there are 14 themes. These themes were divided into three categories, namely **Process vs Product** (to determine those attributes of process that most influence attributes of the product), **Process vs Process** (to determine the inter-relationship between various attributes of the process) and **Product vs Product** (to determine the inter-relationship between various attributes of the product). They are :

PROCESS VS PRODUCT

- Approach to developing high-quality - low defects software
- Coding reflects on software size and functionality
- Results of unrealistic project scope
- Quality of students' effort reflects the quality of the final product

PROCESS VS PROCESS

- Students and staff adviser relationship
- How teams choose to spend their time
- Effective team effort and good project management
- Drawbacks of working alone in a group project
- Importance of selecting the right development tools
- Usefulness of using a methodology
- Negative impact of older students in a group project environment
- Drawbacks of mixed male/female project groups
- Productivity of students reflects on coding

PRODUCT VS PRODUCT

- Judging functionality and quality of undergraduate software projects

PROCESS VS PRODUCT

10.3.1 Approach To Developing High-Quality - Low Defects Software

(a) Testing Time vs Solution Quality (0.71)
Testing Time (%) vs Solution Quality (0.71)
Design Time (%) vs Defects Found (-0.59)

The results showed that the teams who spent more time on software design and testing, tended to produce better quality software with less defects.

By spending more time on design, the teams would be able to define a better solution to the problem. Vliet suggests (1993, p. 171) that a good design is a major factor in developing a successful product. Vliet (1993, p. 171) postulated that a "well-designed system is easy to implement, is understandable and reliable, and allows for smooth evolution". Conversely, badly designed systems are harder to maintain, difficult to test and are less reliable. The design phase can be regarded as one of the most crucial phases in the software development life-cycle.

More time spent on conducting software testing should enable students to locate and remove major errors and bugs, thereby producing a better quality software product. During the construction of software, many errors are bound to be made. Locating and fixing these errors through excessive testing is a very time consuming activity and it is fair to say that not all errors will be found (Vliet, 1993, p. 315). Vliet suggests (1993, p. 315) that to have good testing is as difficult as having a good design.

This has been supported by the results that have been obtained. However, it would be reasonable to expect the results to show that extensive testing results in the software having fewer defects. This relationship was not evidenced in the analysis. This may well be due to time constraints as testing was only conducted against the software design and coding and not on the software's logical constraints and business functionality.

(b) Solution Quality vs Total Hours Spent (0.58)
Solution Quality vs Team Work (0.62)
Solution Quality vs Contribution To Project (0.51)

The results revealed that teams which spent more time on their project received higher marks for software quality. It also showed that if the students were satisfied with their contribution to the project and worked as a team, they also tended to produce better quality software.

Students who spent more time together on group activities should invariably exhibit high team morale. Working well together also implied that team interaction was high. When team interaction was high, students would have an opportunity to maximise their contribution to the project. This tended to create an environment where there was no dominant individual controlling the team. Allowing every student to have his or her say in the project improved the flow of ideas and suggestions. Having obtained a wider range of ideas and suggestions, the students could then select those that were applicable to their problem. Students who were able to produce a well-defined solution invariably produced a better quality software.

(c) On Schedule vs Defects Found (-0.80)

This result showed that those projects that were on schedule tended to have less defects.

The result suggested that teams who completed their project on time dedicated sufficient time to 'proper software testing. It also implied that the students did not have to rush to complete their software. Facing schedule pressure often results in poor product quality (Gilb, 1988, p. 326). Sufficient testing and time to complete the project were some of the important factors that led to the development of software with fewer defects. Running behind schedule usually has disastrous effects on the project. Jones (1991, p. 226) said that sometimes schedule pressure can actually have some positive impact on the team morale. Jones (1991, p. 226) also pointed out that excessive and unrealistic schedules are probably one of the most "destructive influences in all of software". Jones (1991, p. 226) stated that unrealistic schedules not only tend to cause the projects to fail but they also "cause extraordinarily high voluntary turnover among staff members".

10.3.2 Coding Reflects On Software Size And Functionality

(a) Productivity Rate vs Solution Functionality (0.57)
Productivity Rate vs Software Size (0.79)
Productivity Rate vs Software Usability (0.66)
Coding Time vs Solution Functionality (0.52)
Coding Time vs Software Size (0.90)

The results showed that the teams who were more productive and spent more time on coding, tended to produce a larger piece of software with greater perceived functionality. It also showed that the teams who were more productive, tended to produce a piece of software with higher usability.

It was apparent that the amount of time that was spent on coding directly impacted on the size of the software but not necessarily on its proposed functionalities. This implies that to produce a larger piece of software, the productivity rate of the students would have to be high. It is fair to conclude that the productivity rate of students and the time spent on coding are inter-related, as shown in **Section 10.3.13**.

10.3.3 Results Of Unrealistic Project Scope

(a) Analysis Time (%) vs Software Size (-0.66)
Analysis Time (%) vs Productivity Rate (-0.63)[19]
Requirement Time (%) vs Software Size (-0.56)
Requirement Time (%) vs Productivity Rate (-0.65)
Requirement Time (%) vs Solution Quality (-0.58)
Requirement Time (%) vs Usability (-0.65)

The results reflected that the students who spent a larger proportion of their time on analysis and requirement, would produce a smaller piece of software that was lacking in quality and usability. The supporting results also indicated that the students who spent a larger proportion of their time on analysis and requirement were less productive.

The results seem to be unusual. The data presented in **Sections 4.2.3** and **4.2.5**, implied that the teams generally were not very satisfied with their staff adviser's advice and their client. The information presented in **Section 4.3**, indicates that the majority of the staff advisers were inexperienced and therefore, of little benefit to the students. Accordingly it was apparent that not all of the students were properly supervised and the requirements were not clearly defined by the client. Furthermore, the majority of the students were new in the area of software development and it was reasonable to say that these students would not have had sufficient experience to realistically define their scope. Bearing all this in mind, the analysis and requirement phases were conducted during the first semester of the project. The coding phase then commenced in second semester. It was apparent that

[19] **NOTE** : Although Analysis Time (%) and Productivity Rate are both process metric, they are represented here as supporting results. All supporting results will be represented in italic.

the students then realised that their scope was unrealistic and they were unable to handle the situation. Confrönted with fear and confusion, and having no positive supervision, the majority of students were left to their own devices. Hence this led to the negative effects on productivity, software size, quality and software usability.

According to Pressman (1992, p. 68), the scope describes the "function, performance, constraints, interfaces and reliability" of the software. With this project, the students were more concerned with the functions and interfaces of the software. Unfortunately, these two requirements were not clearly defined by the client. Hence the students had no choice but to define their own set of functions and interfaces. As a result, some groups fell into the trap of over-defining the scope.

It is recommended that staff members with sufficient background in software engineering should be assigned to be staff advisers. This would be fair to the students as they would be able to benefit and learn from their staff adviser's advice. A further recommendation is that students should have more access to the client. That would allow the client's requirements to be easily gathered and verified.

10.3.4 Drawbacks Of Mixed Male/Female Project Groups

(a) Female Students (%) vs Defects Found (0.76)

The result showed that the software would have more defects if there were more female students in the group.

This result requires some interpretation. The groups were primarily male in make-up. When students were working in group, some sort of a bond tended to be established between them. Also when working closely together, members of the opposite sex would tend to be hesitant about voicing out the problems and errors found during the development of the software. They might worry about hurting the other member's feelings or ego.

Generally, when a group was comprised entirely of male students, they tended to be more frank and open toward one another. But when dealing with a member of the opposite sex, the males tended to be more polite and less aggressive. Helen Marshall (1987, p. 45) proposed that male students were more active and more confident in debates, whereas female students were more self-conscious when talking in public. It would seem that the inability to express one's opinion outrightly towards a member of the opposite sex was the cause of this negative impact.

One other possible reason may lie with the mind set of today's society. We are now moving towards the twenty-first century and many have come to believe and accept the right of equal opportunities for both men and women. But, when it comes down to more technical matters, men still believe that they are better at handling the situation. As Marshall (1987, p. 111) said, "many people still feel, for example, that males should take lead in activities and projects, and that females shouldn't be 'pushy'". Marshall (1987, p. 42) also argued that a consideration of the enrolments of tertiary students, would show that there was a heavy concentration of male students in engineering and computer science courses and female students in humanities, social science, librarianship and nursing courses. Assuming that this was the case, women students may not be taking (or invited to take) a more active role in contributing to the project. Women students may ended up only doing clerical activities. If this was the case, the group would only be partially utilising their human resources, hence leading to this negative outcome. Unfortunately, there were no all-female project teams on which to further develop this hypothesis.

10.3.5 Quality Of Students' Effort Reflects The Quality Of The Final Product

(a) Cross Scores vs Solution Functionality (0.66)
Cross Scores vs Solution Quality (0.81)
Cross Scores vs Usability (0.59)
Course Averages vs Usability (0.60)

The results disclosed that high cross scores were given by teams whose software was perceived by the judging panel as representing a good product. They also indicate that teams with higher course averages tended to produce software with better usability.

The score for peer-assessment was awarded by each student and was based on their perception of the overall performance of each member. It seems to suggest that students who awarded each other high marks were also happy working as a team. This in itself implies that the students were satisfied with all the various aspects of how their project team was managed. It was fair to say that teams that felt good about their own performance were sure that they had developed the software well. The results above support the conclusion that there was a positive relationship between the marks allocated within a group and the marks awarded by the judging panel for software functionality and quality. This was further supported by the usability score that was derived. Teams having higher course averages also tended to produce software with high usability. This implied the brighter and harder working students did have a positive impact on the overall project.

It is recommended that if similar research is to be conducted in the future, the students' scores from the programming units and units where the students are required to work as a team should be used, rather than just the students' course averages. Using these proposed scores can provide an insight into individual students' programming skills and team interaction.

PROCESS VS PROCESS

10.3.6 Students And Staff Adviser Relationship

(a) Staff Adviser's Advice vs Team Work (0.72)
Staff Adviser's Advice vs Contribution To Project (0.62)
Staff Adviser's Advice vs Project Management (0.67)

The results displayed that if the students were satisfied with their staff adviser's advice, they were also satisfied with their contribution to the project, working as a team and the way the project was managed.

Teams having valued feedback from their staff advisers seemed to work better as a team, value each other's contributions and were happier with the management of the project. If students felt they were being well-directed, they were happier with the way the team was working.

10.3.7 How Teams Choose To Spend Their Time

(a) Team Size vs Total Hours Spent (0.66)

The result evidenced that if a team were to have more students, they tended to spend more time on the project. The data seemed to suggest that larger teams would have more man-hours to devote to the various tasks. In smaller teams, the students did not have the luxury of performing some tasks as thoroughly as they would have liked. In certain cases, smaller teams might have had to compromise certain activities such as software testing.

(b) Team Size vs Hours Spent In Testing (0.66)

The result exhibited that if a team had more students, they tended to spend more time on software testing.

The observation made was that for larger teams, the students would have enough human resources to spare for conducting extensive software testing. Unfortunately, in cases where there were only three students in a group, each student would have to perform in the majority of the tasks. With no one to delegate the tasks to, a team of three students would have to work twice as hard compared to a team of six students. By almost doubling the work load and having to meet an inflexible schedule, the small team would choose to sacrifice the time on testing over time allocated for coding.

According to Shneiderman (1980, p. 129), some social psychological research suggests that members of small groups tend to encourage each other to perform better because they feel that the group members will "recognise good work and criticise poor performance". Unfortunately, small groups are also most likely to be affected by anxiety and fear of failure. Teams should not be allowed to get down to a small size. A group of five students is a reasonable size and is recommended for future undergraduate software engineering projects.

(c) Requirement Time vs Analysis Time (0.66)
Requirement Time (%) vs Analysis Time (%) (0.65)

The results revealed that the teams who spent more time gathering requirements, would also spend more time analysing these requirements.

Most of the students spent a fair amount of time conducting their research into meeting the requirements of the project. They were able to obtain information from orchardists, the Taxation Department and the Weather Bureau. Having obtained this information, the students also spent a large proportion of their time analysing the information.

(d) Total Hours Spent vs Design Time (%) (0.56)
Total Hours Spent vs Coding Time (%) (-0.81)
Total Hours Spent vs Testing Time (%) (0.73)

The results indicated that if the teams had more time to spend, they would spend it on design and testing, and not on coding.

Expending more time on the design phase would lead teams to define a better solution to the problem. By spending more time on software testing, the students were able to locate and remove potential errors and bugs. If the students were able to define a proper solution and perform sufficient testing, in the long run, the software would require less re-coding and modification. However, the amount of time spent on testing depends greatly on the amount of spare time the group has before meeting the deadline. In cases where projects were behind schedule, software testing was often compromised (Paulk et al. 1993, p. 2).

(e) Coding Time (%) vs Analysis Time (-0.98)
Coding Time (%) vs Analysis Time (%) (-0.58)
Coding Time (%) vs Design Time (-0.77)
Coding Time (%) vs Design Time (%) (-0.73)

These results showed that the teams who spent more time on analysis and design, would subsequently spend less time on coding.

The data presented here provides further support to the point made in the previous section (**Section (d)**). The students could better understand the problem through extensive analysis and derive a better solution through extensive design. If the students were clear on what to develop and how to develop it, they were most likely to build the right software the first time around. If this were the case, the software would require less re-coding and modification.

10.3.8 <u>Effective Team Effort And Good Project Management</u>

(a) Total Hours Spent vs Team Work (0.61)
Analysis Time vs Team Work (0.66)
Analysis Time vs Contribution To Project (0.57)
Design Time vs Team Work (0.55)
Design Time vs Contribution To Project (0.58)
Design Time (%) vs Contribution To Project (0.60)

The results showed that if the teams had more time to spend, they would spend it on team activities. It also showed that the students who spent more time on analysis and design, were also satisfied with their contribution to the project and team work.

It was important that every student worked as part of the team, and contributed to the project whenever possible. Being a team project, every students' opinions and suggestions should be heard. Whenever possible, the students should function as a team. The data suggests that teams who spend more time on team activities like analysis and design were also satisfied with their individual contribution to the project and team work. This implied that the students were functioning as a team during the analysis and design phase, which was not unexpected.

(b) Testing Time vs Team Work (0.58)
Testing Time (%) vs Team Work (0.55)
Testing Time vs Cross Scores (0.85)
Testing Time (%) vs Cross Scores (0.68)

These results evidenced that the teams who spent more time on software testing, were more satisfied with their team work and gave each other a good score during the peer-assessment.

In most cases, the software coder(s) would be different from the software tester(s). This result indicates the involvement of team effort. Hence, it is fair to say that testing is good for team spirit. The fact that the students score each other highly for the peer-assessment suggested that students gave good marks to each other when they saw effort in testing.

(c) Project Management vs Team Work (0.79)
Project Management vs Contribution To Project (0.92)
Team Work vs Contribution To Project (0.87)
Total Hours Spent vs Cross Scores (0.72)

These results indicate a positive correlation between satisfaction with the project management, satisfaction that individual contributions were recognised and satisfaction with the way the team worked together. In addition, they also show that if the students had more time to spend on the project, they tended to award each other a higher score for the peer-assessment.

Sommerville (1989, p. 24) said that the project leader must understand the characteristics of his or her team members and understand how these individuals worked together. A well-managed project provided an environment where team members were well accepted by their peers and their contributions appreciated. A group that worked well together implied that every student was able to participate in the development process. Cases where the students spent more time working on the project, suggested evidence of team involvement thereby leading to the high peer-assessment score. It is believed that the students awarded the peer-assessment score based on their hours together working as a team.

10.3.9 Drawbacks Of Working Alone In A Group Project

(a) Coding Time vs Contribution To Project (-0.54)
Coding Time (%) vs Contribution To Project (-0.63)
Coding Time (%) vs Team Work (-0.69)

The results reflected that the teams who spent a larger proportion of their time on coding, were less satisfied with their contribution to the project and team work. It tended to suggests that too much time spent coding is not good for team spirit.

In most cases, especially with students, the coder(s) tended to work independently from the team. The coder(s) would develop the software according to the design specifications without having input from the rest of the team. This suggested that there was not much team effort involved and not every student had a say on how the software was to be coded. This argument is supported by the results presented above. It suggests that the remaining team members were not very satisfied when someone from their team worked alone.

It would be ideal if the students were able to developed an egoless programming environment to work in. Sommerville (1989, p. 37) defines egoless programming as "a style of project group working which considers programs to be common property and responsibility of the entire programming group irrespective of which individual group member was responsible for their production". Weinberg (cited in Sommerville, 1989, p. 37) suggests that by making the production of a program a group effort, rather than an individual effort, creates a good working environment. To support the views expressed, Sommerviile (1989, p. 38) pointed out that programmers who wrote the program tended to defend that program against criticism. That defensiveness tended to work against good team spirit.

(b) Staff Adviser's Advice vs Coding Time (%) (-0.67)
Staff Adviser's Advice vs Analysis Time (0.61)

Teams who were happy with their staff adviser's advice spent more time on analysis and less time on coding - reflecting the advice given.

Staff advisers tended to advise spending time on analysis and design rather than coding. This advice seems to have been taken.

(c) Coding Time (%) vs Project Management (-0.60)

The results showed that the students were less satisfied with the way their project was managed, if they spent a larger proportion of their time on coding.

The role of the project leader was to oversee all the project related activities. However in the situation where the coder worked alone, even the project leader had very little influence over the coding process. This was of course reflected by the result presented above. However, the results further suggest that teams who were concerned about the way the project was managed, made up for it by spending more time on coding.

10.3.10 Importance Of Selecting The Right Development Tools

(a) Development Software Used vs Project Management (0.64)
Development Software Used vs Team Work (0.65)
Development Software Used vs Contribution To Project (0.57)

These results exhibited that satisfaction with the choice of software led to satisfaction with the way the team operated.

Through good project management techniques, the students were able to select the right development tools. The selection process was not performed by the project leader alone. It was a process that involved the whole team. Students were only able to make an objective selection after thorough discussion and weighing the pros and cons of a particular development tool (ie. biased by any sales pitches).

When faced with a deadline, the task of selecting the right development tool would become very important. This was particularly true in a university environment. If the students were to select the wrong development tools, they might be required to spend more time understanding them. This stress and pressure could lead to poor team morale and could reduce team efficiency.

Sommerville (1989, p. 33) said that the "programming ability is language independent and programming language knowledge is held in a representation-independent way". This means that a programmer who is familiar with one programming language will find it relatively easy to learn a new programming language of the same type. All that is required by the programmer is to learn the new syntax because the underlying concepts are the same. However, Sommerville (1989, p. 33) also pointed out that this is only true if the semantic concepts are the same. For example, a programmer who is experienced in structured programming languages (eg. Pascal) may find it difficult to grasp the programming concepts of object-oriented programming languages (eg. Smalltalk) or functional programming languages (eg. Prolog).

The programming foundation for most of the students was based mainly on structured programming languages such as Pascal. The development tools used for this project were all 4GL-type tools which represented a new paradigm to these students. Therefore, the students were required to spend more time understanding this paradigm before they could apply it to their project.

(b) Design Time (%) vs Development Software Used (0.71)

The result displayed that the students who spend a larger proportion of their time on design, were also satisfied with the development software used.

During the design phase, the students would have known what was required of the proposed software. They would have figured out what was required to develop the software. From this, the students would have an idea of the type of development tools that they required. This knowledge would most certainly assist them in selecting the right commercial development tools that were available on the market.

10.3.11 Usefulness Of Using A Methodology

(a) APT Methodology vs On Schedule (0.61)
APT Methodology vs Project Management (0.59)

The results showed that adherence to the APT methodology led to the project being on schedule and general satisfaction with the project management.

The students were taught about the importance of having a good development methodology. The department ensured that what was being taught was also being practised. Hence the students were encouraged to use the APT methodology (EXECOM, 1991). The project leader that followed the guidelines of the methodology was able to better prepare the tasks and activities that needed to be performed, and were also able to set up realistic project milestones. Projects that were able to meet these milestones were more likely to be completed on schedule.

10.3.12 Negative Impact Of Older Student(s) In A Group Project Environment

(a) Average Age vs Project Management (-0.62)
Average Age vs Team Work (-0.62)
Average Age vs Contribution To Project (-0.53)
Average Age vs Staff Adviser's Advice (-0.77)
Average Age vs Testing Time (%) (-0.56)
Average Age vs Usability (-0.55)

These results indicated that if the students were older, they tended to be less satisfied with their project management, team work, contribution to the project and staff adviser's advice. It also showed that the older students would allocate a smaller proportion of their time to software testing and tended to produce software with lower usability.

The results presented above suggest that having older students working in a group project has some negative impact on the project. There are three possible explanations.

Firstly, it was very common that the older member of the team would get elected as project leader. The older students tended to have experience from another discipline, and little or no experience in the art of managing the software project. Due to this lack of experience, such project leaders might not be able to effectively command the group and the project. Pressman (1992, p. 42) states that for a project to succeed, management must enforce good project management practices. He further added that it would be expected that all project leaders understand how to do it, unfortunately, many do not. Pressman was referring to a real world situation, which also holds true to a university environment.

Secondly, older students tended to be more cynical about things and were less enthusiastic than their younger team mates. It might be the case that the older student had experienced similar projects before and found the current project less challenging or too trivial. This may have resulted in them being less active or uninterested in group activities. Older students might also be reluctant or too proud to take advice offered by their younger team members and staff advisers.

Thirdly, some of the older students might be on a career change and were unable to cope with the paradigm shift. What they might have learnt from past experiences might not be applicable to the current situation. For example, the testing skills that they acquired from past experiences might be inapplicable to testing a piece of software.

10.3.13 Productivity Of Students Reflects On Coding

(a) Productivity Rate vs Coding Time (0.82)
Productivity Rate vs Coding Time (%) (0.71)

The results revealed that the delivery rate of function points was higher for groups that spent more time on coding.

Where teams spent more time on coding, they tended to produce a larger piece of software, as supported in **Section 10.3.2 (a)**. If the teams were able to produce a larger piece of software within the allocated time, it is fair to concluded that the teams were also delivering function points at a faster rate.

PRODUCT VS PRODUCT

10.3.14 Judging Functionality And Quality Of Undergraduate Software Projects

(a) Solution Functionality vs Software Size (0.53)
Solution Quality vs Usability (0.72)

The results showed that if the software was high in functionality and quality, as perceived by the judging panel, they would also have larger size and better usability.

The score for solution functionality and solution quality was awarded by the judging panel during the demonstration of the software. It was very likely that big pieces of software would provide more functionality. This was one of the criteria used by the judging panel. The judging panel awarded the score for solution functionality based on their perception of the size of the software. Based on the result, it was fair to say that the judging panel's perception was fairly accurate. The solution quality was also awarded by the judging panel based on the perceived quality of the functions provided by the software.

Usability was one of the quality criteria used by the panel, so a correlation with tested usability is not-surprising.

Though the approach adopted by the judging panel appears to prove useful and effective, it is recommended a more objective approach to this matter be adopted. The size of a piece of software might reflect on the software's functionality but this functionality does not necessarily address the requirements of the client. It is proposed that the judging panel prepare a task list based on the client's requirements. The score could then be awarded based on the number of requirements that each piece of software met. It is considered that this would be a fairer approach. It would be ideal if the judging panel was able to judge each piece of software based on the other product attributes such as reliability, portability, etc. Unfortunately, due to the time constraint, judging the software's functionality and quality would have to suffice.

10.4 DIFFICULTIES ENCOUNTERED DURING ANALYSIS

To work on a research project such as this, it is necessary to be extra careful on selecting the right method(s) of collecting raw data. The achievement of a successful study, depends on the quality of the data collected. As Fenton (1991, p. 115) said, "data collection is the kernel of any measurement programme". If the data collected was unrealistic, incomplete or inconsistent, it would produce results that would be meaningless or inconclusive.

During the course of collecting data, there were a series of obstacles. It is believed, no other university in Australia has conducted such an exercise. Hence, there were no guidelines to follow and there was a lot of uncertainty as to the approach of data collecting.

What may be applicable in the industry may not necessarily be applicable in a university environment. For instance, the .students that took part in this research project did it out of goodwill. They were not paid for their effort and were not forced to participate. Unfortunately, the data collected during the course of the project were mainly incomplete and inaccurate. To overcome this problem, a final set of questionnaires was prepared and given to the students after their project demonstration. It was made mandatory for all the students to fill in the questionnaire. From the final set of questionnaires, all the necessary data was collected from the students. Therefore, the data was more complete and consistent. This final set of data has been the backbone to this entire research. From this experience, it is patently obvious that to collect a more complete set of data, it should be made mandatory for the students to participate under a controlled environment. However, in doing so, the students must be informed that the results of the research would not be used against them.

In total, there were 15 pieces of software of which only 10 were found to be functioning, even though all the software appeared to be functioning during the demonstration. Since it was not mandatory for the students to submit their software for evaluation, it was concluded that the students failed to provide their current and working model. If all 15 pieces of software were found to be working, it would greatly improve the results that were derived.

As part of this research, it was required to perform some software metrics on the software. The most notable one is Albrecht's Function Point Analysis method. To gather more current information on counting function points, the Australian Software Metrics Association (ASMA) were written to requesting more information. After almost a month, the ASMA replied saying that they were unable to release any information due to copyright reasons. Being an organisation that should be encouraging the measurement of software, the service that they offered was less than encouraging. Since the organisation depends heavily on volunteer workers, it is only fair to say that they might not have the human resources to deal with general enquires in great length. The International Function Point User Group (IFPUG), in the United States, were also written to requesting similar information. Unfortunately, they have yet to respond. As a result, it was unavoidable to use an older version of rules on counting function points based on Dr Rudolph's (1989) seminar paper.

10.5 CONCLUSION

After careful analysis of all the data gathered, a lot of factors that lead to a good software development environment become apparent. Though some may already have been well known, there were others that were unique to a university environment. To address the questions raised by this research project, the following conclusions have been reached.

It is now evident that having a staff adviser assigned to supervise the project group has its advantages. With tighter supervision, the staff adviser would be more aware of the progress of the group. Opinion and supervision from the staff adviser could help students guide their project towards the right direction and promote team work.

The research results showed that if students were to spend more time on the requirement, analysis and design phase, and conducting extensive software testing, they would produce better quality software. It revealed that the software would also require less re-coding and modification, having fewer defects and have better software quality. However, great team effort is required in order to deliver a high quality software. Every student's contribution must be considered. The results also indicated that the groups that were able to deliver their software on time had fewer defects. This implied that the groups that were on schedule had more time to conduct proper software testing. In general, students that were satisfied with all the aspects by which their project was handled and conducted, tended to produce software that had better functionality, quality and usability.

Project management has always been one of the key factors in the success of a project. The same principle applies to a university environment. The research evidenced that with good project management there was better control over project and the team. The project leader was able to "glue" the team together to form an environment where everyone was able to contribute and participate in all the various activities. This is one of the attributes for making a winning team.

How is a winning team defined? A winning team can be classified as one where the team worked well together, are actively involved in all team activities, have a well managed project and have strong interactions with their staff adviser. A winning team will also realise the importance of a methodology and adhere to it and, carefully and objectively select the right development tools. A winning team may not get the best mark, but the individual students will have gained most from the experience.

However, the research also showed that the software coder(s) tended to work alone. Students have to realise that in the work force, there is no such thing as a lone coder. The coder's work would be constantly monitored by his or her peers. The same should be applied to students. Students should work as a team during the coding phase with input and assistance from the other team members.

Though selecting the right development software may not seem to be a major issue, it is, especially in a university environment. Unlike the real-world, where a project deadline could be modified or postponed, the students were faced with a strict deadline which they had to deliver. Unable to complete the project within the deadline might result in them being penalised academically. That is why selecting the right development software is important. Selecting the wrong software might require the team to spend more time understanding it. The students had to be quite competent in the development software within the time frame of 2 semesters in order to successfully develop the final product.

One pitfall that students frequently fall into is that of defining a scope that is too large. Most of the students were inexperienced in this area and they had a tendency to do this. Most of the time, the group realised too late in the project that they were unable to cover all the areas defined in the scope. This could become a serious problem especially if the groups were poorly supervised and the requirements were not clearly defined by the client or user. The research had shown that students that were unable to recover from this problem were generally less productive and would produce software with less functionality, poor on quality and less usefulness.

The APT methodology (EXECOM, 1991) has been used by this department for the past few years. The results gathered from this study for the first time, has provided the department with some empirical data to support the usefulness of this methodology. Even though the APT methodology was not well received by the students, the research has shown that students who adhered to the APT methodology were able to have better control over the project and in doing so, were able to complete their project on time. These students will be future contributors to the arena of software engineering. If they could apply what they have learnt from this exercise into the work force, this would provide some hope to future software development projects with the likelihood of them being completed on schedule. This is something that every real world developer hopes to achieve on all their projects.

Each group was required to demonstrate their software before a judging panel. The research showed that members of the judging panel where able to successfully and objectively award the appropriate score on the software functionality and quality based on the software's perceived size and usability. It is recommended however that an alternative approach be adopted whereby the students must demonstrate the key features of the software based on a task list provided by the judging panel. Scores could then be awarded based on the number of features that the students developed. This would seem to be a more accurate and objective approach to judging a piece of software.

The research has also shown that having older students working in a group project had its disadvantages. The results showed that older students tend to upset activities such as project management, team work, team contribution, etc. The study suggested that older students from other disciplines should keep an open mind when it comes to developing software. Older students who were elected as project leaders should be less cynical, more enthusiastic and think of the team's welfare. In a group project, every student is affected by the performance of their peers. Older students should be able to take advice and criticism from their peers and staff adviser.

Another result of concern was that mixing male and female students in a project group appears to contribute to the software having more defects. Having members of the opposite sex working together can cause problems in communication. It was very common that a female student would not tell her male team-mate(s) that he was wrong. This was similarly evident with male students as well. Both sexes appeared to be conscious of hurting the other's person feeling or ego. This was also true with overseas students. Unfortunately, reservation of one's opinion may jeopardise a project.

In conclusion, no claim is made or remotely suggested that the research gathered is 100 per cent accurate and without errors. If this research is published, errors from the study should be corrected by subsequent researchers. It is hoped that if the results concluded are later found to be incorrect, "its publication will be at least a step towards new and correct data that will benefit the software industry (Jones, 1991, p. 125)" and learning institutions. After all, "the industry cannot proceed into the twenty-first century with no quantitative data at all ... (Jones, 1991, p. 125)".

10.5.1 RECOMMEDATIONS TO PROJECT CO-ORDINATOR

Below are the recommendations as a result of the study.

- Collection of project data should be made mandatory.
- Staff advisers should have a reasonable amount of knowledge regarding the standard software development methodology adopted by the Computer Science department.
- Staff advisers should be interested and volunteer for the role.
- Staff advisers should have sufficient knowledge of the software development process.
- Staff advisers should have a clear understanding of the nature of the software engineering project.
- Team supervision should be more consistent.
- Teams reporting to their staff adviser should be made mandatory.
- Teams should use the same development software.
- Teams should adhere to the development methodology when possible.
- Procedures for gathering system requirements should be improved.
- Software produced by each team should be assessed based on a representative task list which describes the client's requirements.
- Scores from students' programming units and units where the students are required to work as team should also be used as criteria for team formation.

All of these recommendations have been taken on board by the software engineering project co-ordinator for 1994.

BIBLIOGRAPHY

Adams, E. J. (1993A Project-Intensive software design course. SIGCSE Bulletin, 25 (1), 112 - 116.

Alavi, M. (1984, June). An Assessment Of The Prototyping Approach To Information Systems Development. Communication Of The ACM, 27 (6), 556 - 563.

Alavi, M. & Wetherbe, J. C. (1991, May). Mixing Prototyping And Data Modelling For Information System Design. IEEE Software, 8 (3), 86 - 91.

Albrecht, A. J. & Gaffney, J. E. Jr. (1983, November). Software Function, Source Lines of Code, and Development Effort Prediction : A Software Science Validation. IEEE Transactions On Software Engineering, SE9 (6), 639 - 648.

Australian Software Metrics Association, The. (1993a, February). Project Database - Collection Package, Release 2. Victoria : Australia.

Australian Software Metrics Association, The. (1993b, September). Project Database, Release 3. Victoria : Australia.

Baker, M. D. (1991, May 20 - 24). Implementing an initial software metrics program. Proceedings of the IEEE 1991 National Aerospace and Electronics Conference NAECON 1991, pp. 1289 - 1294.

Basili, V. R. & Weiss, D. M. (1984, November). A Methodology for Collecting Valid Software Engineering Data. IEEE Transaction On Software Engineering, SE10 (6), 728 - 738.

Behrens, C. A. (1983, November). Measuring the Productivity of Computer Systems Development Activities with Function Points. IEEE Transaction On Software Engineering, SE9 (6), 648 - 652.

Boehm, B. W. (1981). Software Engineering Economics. New Jersey : Prentice-Hall, Inc.

Boehm, B. W. (1984, January). Software Engineering Economics. IEEE Transaction On Software Engineering, 10 (1), 4 - 21.

Boehm, B. W. (1988). A Spiral Model of software development and enhancement. IEEE Computer, 21 (5), 61 - 72.

Briggs, J. (1991). Group projects in software engineering at York. SIGCSE Bulletin, 23 (4), 48 - 50.

Calliss, F. W. & Trantina, D. L. (1991, October 7 - 8). A controlled software maintenance project. Software Engineering Education SEI Conference 1991 Proceedings, pp. 25 - 32.

Clapp, J. (1993). Getting started on software metrics. IEEE Software, 10 (1), 108 - 111.

Curtis, B., Sheppard, S. B., Milliman, P., Borst, M. A. & Love, T. (1979, March). Measuring the psychological complexity of software maintenance tasks with Halstead and McCabe Metrics. IEEE Transaction on Software Engineering, pp. 96 - 104.

Davis, D. B. (1992). Develop applications on time, every time. Datamation, 38 (22), 85 - 89.

EXECOM (1991). Student APT Methodology. Western Australia, Australia.

Felican, L. & Zalateu, G. (1989, December). Validating Halstead's Theory for Pascal Programs. IEEE Transaction On Software Engineering, 15 (12), 1630 - 1632.

Fenton, N. E. (1991). Software Metrics : A rigorous approach. Norwich : Page Bros Ltd.

Ferens, D. V. & Gurner, R. B. (1992, May 18 - 22). An evaluation of three function point models for estimation of software effort. Proceedings of the IEEE 1992 National Aerospace and Electronics Conference, pp. 635 - 642.

Ferrari, D. (1986, June). Considerations on the Insularity of Performance Evaluation. IEEE Transactions On Software Engineering, 12 (6), 678 - 683.

Freund, J. E. & Simon, G. A. (1992). Modern Elementary Statistics. New Jersey : Prentice-Hall, Inc.

Gilb, T. (1988). Principles of software engineering management. Avon : The Bath Press.

Graham, C. L. & Jeffery, D. R. (1990, January). Function Points in the Estimation and Evaluation of the Software Process. IEEE Transactions On Software Engineering, 16 (1), 64 - 71.

Grant, D. D. & Smith, R. (1991). Undergraduate Software Engineering - An Innovative Degree at Swinburne. The Australian Computer Journal, 24 (3), 106 - 114.

Grubb, P. A. (1991, October 22). Undergraduate Software Engineering Projects - Keeping the momentum going. IEE Colloquium On 'Teaching Of Software Engineering - Progress Reports, pp. 5/1 - 5/3.

Grupe, F. H. & Clevenger, D. F. (1991). Using function point analysis as a software development tool. Journal of Systems Management, 42 (12), 23 - 26.

Heemstra, F. J. & Kusters, R. J. (1991). Function Point Analysis : Evaluation Of A Software Cost Estimation Model. European Journal Of Information Systems, 1 (4), 229 - 237.

Ince, D. (1990, May). Software Metrics : Introduction. Information And Software Technology, 32 (4), 297 - 303.

Jones, C. (1991). Applied software measurement : assuring productivity and quality. New York : McGraw-Hill, Inc.

Kemerer, C. F. (1987, May). An empirical validation of software cost estimation models. Communication Of The ACM, 30 (5), 416 - 429.

Kemerer, C. F. (1993). Reliability of Function Points measurements : A Field Experiment. Communication Of The ACM, 36 (2), 85 - 97.

Keyes, J. (1992). New metrics needed for new generation : lines of code, function points won't do at the dawn of the graphical, object era. Software Magazine, 12 (6), 42 - 50.

Kitchenham, B. A. (1992, April). Empirical studies of assumptions that underlie software cost-estimation models. Information And Software Technology, 34 (4), 211 - 219.

Kizior, R. J. (1993). Function Point Analysis : A Primer. Interface : The Computer Education Quarterly, 15 (1), 42 - 49.

Kusters, R. J., Genuchten, M. J. I. M. & Heemstra, F. J. (1990, April). Are software cost-estimation models accurate? Information And Software Technology, 32 (3), 187 - 190.

Marciniak, J. J. & Reifer, D. J. (1990). Software Acquisition Management : managing the acquisition of custom software systems. Canada : John Wiley & Sons, Inc.

Marshall, H. (1987). Sex, gender and society. Melbourne : RMIT Ltd.

Mills, E. E. (1988, December). Software Metrics - SEI Curriculum Module SEI-CM-12-1.1. Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

O'Brien, S. J. & Jones, D. A. (1993). Function Points In SSADM. Software Quality Journal, 2 (1), 1-11.

Orchard project bears fruit. (1993, December). Edith Cowan University Digest, p.8.

Paulk, M. C., Curtis, B., Chrissis, M. B. & Weber, C. V. (1993, February). Capability Maturity Model for Software CMU/SEI-93-TR-24. Technical Report, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Perlis, A., Sayward, F. & Shaw, M. (1981). Software Metrics : an analysis and evaluation. Massachusetts : The MIT Press.

Pressman, R. S. (1987). Software Engineering : A Practitioner's Approach. New York : McGraw-Hill, Inc.

Pressman, R. S. (1992). Software Engineering : A Practitioner's Approach. New York : McGraw-Hill, Inc.

Rudolph, E. Dr. (1989, April). Accounting For Software Development : An in-depth guide to the function point analysis. Paper presented at a three day seminar.

Shaw, M. & Tomayko, J. E. (1991, October 7 - 8). Models for undergraduate project courses in Software Engineering. Software Engineering Education SEI Conference 1991 Proceedings, pp. 33 - 71.

Shepperd, M. (1988, March). A critique of cyclomatic complexity as a software metric. Software Engineering Journal, pp. 30 - 36.

Shepperd, M. (1990, May). Early life-cycle metrics and software quality models. Information And Software Technology, 32 (4), 311 - 316.

Shneiderman, B. (1980). Software Psychology : Human Factors in Computer and Information Systems. Boston : Little, Brown and Company, Inc.

Sommerville, I. (1989). Software Engineering, Avon : The Bath Press.

Symons, C. R. (1988, January). Function Point Analysis : Difficulties and Improvements. IEEE Transaction On Software Engineering, 14 (1), 2 - 11.

Symons, C. (1992, July 30). Management : measure for measure's sake. Computer Weekly, pp. 16.

Tate, G. (1990, May). Prototyping : helping to build the right software. Information And Software Technology, 32 (4), 237 - 244.

Vliet, J. C. van. (1993). Software Engineering : Principles And Practice. Chichester : John Wiley & Sons Ltd.

APPENDIX A : EVALUATION REPORT

This section presents the list of errors found during the evaluation of the software.

NOTE : Terms such as application and system are used. In this context, the term ***application*** refers to the application developed by the students, and the term ***system*** refers to the language or application development tool from which the application was developed and consequently executed.

- ❑ **Group 1**
  - ❑ The lookup table did not immediately update the logical file after a new record was added. To have access to the newly created record it was necessary to exit that form first and then go back into it.
  - ❑ When trying to create a new MARKET record, an error occurs causing the operation to halt. The system reported that a macro for FAX NUMBER could not be found.
  - ❑ Unable to create a new BLOCK & ROW record.
  - ❑ Unable to create a new SHED TICKET record.

- ❑ **Group 2**
  - ❑ The application did not check for *beginning* and *end-of-file* error, which caused the system to halt.
  - ❑ The application did not check for *out-of-bound* errors, which caused the system to halt.
  - ❑ The application had checking mechanisms for duplicate records. However, during testing, the system behaved unstablely when duplicate records were found.
  - ❑ The application was unable to create and delete records from the FRUIT module.
  - ❑ The remaining modules were able to create and update records but were unable to delete records.

  - In the PURCHASE module, the *discount* field truncates all values with decimal points. (For example : 0.10% is truncated to 0.00%).
  - The SAVE RECORD from the pull-down menu was redundant.
  - The SUPPLY INVOICE REPORT was not available.
  - The MOST PROFITABLE REPORT was not available.
  - The ITEMS/ASSETS ORDERS REPORT was not available.

- **Group 3**
  - The application did not have any help options, except those from the system.
  - The form design was done poorly, ie. inconsistent fields tab and inability to distinguish between fields that could be edited and those that could not.
  - Tree Lookup Table was not available.
  - The FRUIT PICKING and SALES modules were not available.
  - The TREE PLANTING module was not able to create, delete or update any records.
  - In the WORK DETAIL module, before a record was deleted, the application prompts for confirmation for approximately a dozen times. This module was also unable to create or update any records.
  - The SPRAY module was unable to delete any records.
  - The STOCK SUPPLIES module was unable to delete any records.
  - The PURCHASE ORDER module was unable to delete any records.
  - The SHIPMENT module was unable to delete any records.
  - It would appear that all modules that require cascaded-deletion were not functioning.
  - The PAYROLL sub-module causes the system to **lock-up**.

- **Group 4**
  - The application had poor screen design, ie. inconsistent fields tab.
  - Most of the options from the pull-down menu did not work or were not available.
  - BLOCK and ROW record cannot be deleted.
  - In the BLOCK and ROW module, the *block number* cannot be selected using the pop-up option provided. To select a *block record*, it was necessary to use the system's "VCR" control buttons.
  - Unable to create SALES ORDER form.
  - In the view and update function of the TREATMENT module, records cannot be selected from the selection list provided. Records can only be selected via the record navigation buttons.
  - The Treatment Effectiveness report was not functioning.
  - Records from the SUPPLIER module cannot be deleted although the option was provided.
  - In certain modules, the create, delete and update options were provided within the form, yet the students had different menu options for these same tasks.
  - Once a record was updated, it does not take effect immediately. To view or access the updated record, it was necessary to exit the form first and then go back in again.
  - The help file was very brief and general. It did not contain instructions on how to use the application.

- **Group 5**
  - The application did not have any help option, except those provided by the system.
  - The screen design did not include *speed bar* or selection buttons. All tasks options had to be selected from a pull-down menu provided by the system.
  - When a new type of tree record was created, this record did not appear in the selection list. To select this newly created record, it was necessary to use the record navigation buttons located on the bottom left corner.

  - If the CANCEL option was selected to abort a process, the application quits.
  - In the EVAPORATION module, it stored the evaporation rate for the whole year (12 months). However, if one of these records was deleted, if did not allow creation of a new record even though the option was provided.
  - In the IRRIGATION module, the "create new record" option was not functioning.

- **Group 6**
  - The application had no help file, although there was a HELP option.
  - The application was unable to create and delete BLOCK records.
  - The application had a very strange method for creating records. It was necessary to select the NEW option first, enter the new data and then select the UPDATE option to store the data onto the file. To create another new record, the form had to be first exited, otherwise the system would generate a *Key Violation* error.
  - The SPRAY details form appeared by itself and cannot be closed.
  - The EMPLOYEE form cannot be opened.
  - There was an error in one of the fields in the HOLIDAY LEAVE form. Once this error was triggered, the EXIT button fails to work resulting in the need to close the form using the Control Menu box located on the top-left corner.
  - The EXIT option in the PAYROLL CONTROL module did not work.
  - The PAYROLL DETAIL module could not create or update any records.
  - The TIMESHEET module could not update any records.
  - There was an error in some of the fields in the EMPLOYEE DEDUCTION / ALLOWANCE form.
  - Some forms kept appearing by themselves and could not be closed.
  - The DELIVERY module was not functioning.
  - The application was unable to delete records from SUPPLIER and FRUITS modules.
  - The FRUIT SALES DETAIL module was not functioning.

    - □ When selecting a REPORT option, the SUPPLIER form came up instead.
    - □ The PRINT PAYSLIPS and CALCULATE PAYS[illegible] modules were not available.
    - □ There was an inconsistency in the record update method. In some cases, records could only be updated after depressing the F9 function key while others do not need to.
    - □ The application was so badly designed that whenever errors were triggered, the form in which the error(s) occurred could not be closed. Usually this would result in having to warm-boot the system.

- □ **Group 7**
    - □ There were no control buttons. All operations had to be selected from the system's pull-down menu.
    - □ The application's help file was too brief and general. It did not provide instructions on how to use the application.
    - □ There was no auto-increment for the PRIMARY KEY field. It was very easy to get into a situation of having duplicate keys - which the application does not allow.
    - □ The records were not indexed or sorted when displayed onto the screen.
    - □ The BLOCK module provides for the addition of a new record but it generates an error when it tries to auto-increment the block number. This problem was overcome by simply putting a unique block number in this field
    - □ The TREATMENT module did not work.
    - □ The application had poor screen design, ie. inconsistent fields tab.
    - □ The WAGES and HARVEST modules were very complicated to use, especially without the aid of the help file or user manual. The option to add WAGES record was not functioning.
    - □ The GROUP CERTIFICATE reports had no report heading.

- **Group 8**
  - The CALCULATE IRRIGATION RATE module was not functioning.
  - In the IRRIGATION module, the update record option was not functioning.
  - The application had very limited functions.

- **Group 9**
  - The application's help file was incomplete. It did not provide instructions on how to use the application.
  - Most of the options which appeared to be available on the application menu were not available.
  - The BUDGET MAINTENANCE module was not functioning.
  - The CHEQUE and CASH PAYMENT reports were not available.
  - Could not create new CONTAINER records if database was empty.
  - The CRATE HIRE module was not functioning and the form could not be closed with the option provided.
  - The FRUIT VARIETY module was not available.
  - The VARIETY PERFORMANCE module was not available.
  - The application was unable to print any reports because the students had hard-coded the printer driver onto the application, hence reducing the portability of the application.
  - The PRINT PREVIEW option was disabled. Therefore, reports could only be printed and not viewed on the screen.
  - The INVOICE, NON-INVOICE PAYMENT, ACCOUNT PAYMENT and OTHER INCOME modules were not available.

- **Group 10**
  - The links between the forms and files were somehow lost during setup. To get the application running, it was necessary to go into the design and re-establish these links. After re-establishing the links, the application still did not function well! It was almost impossible to use!
  - The application did not have any help file.
  - All the reports were not available.
  - The application had very limited functionality. Most of the functions were partially developed or not working correctly.