Implementing flexible software techniques in a 4GL environment

Stephen O'Connor

*Edith Cowan University*

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

Part of the *Software Engineering Commons*

**Recommended Citation**

https://ro.ecu.edu.au/theses_hons/519

This Thesis is posted at Research Online.
https://ro.ecu.edu.au/theses_hons/519
Edith Cowan University

Copyright Warning

You may print or download ONE copy of this document for the purpose of your own research or study.

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

You are reminded of the following:

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

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

• Courts have the power to impose a wide range of civil and criminal sanctions for infringement of copyright, infringement of moral rights and other offences under the Copyright Act 1968 (Cth). Higher penalties may apply, and higher damages may be awarded, for offences and infringements involving the conversion of material into digital or electronic form.
USE OF THESIS

The Use of Thesis statement is not included in this version of the thesis.
IMPLEMENTING FLEXIBLE SOFTWARE
TECHNIQUES IN A 4GL ENVIRONMENT

BY

STEPHEN O’CONNOR BSc (Computer Science)

A thesis submitted in partial fulfilment of the requirement
for the award of

Bachelor of Science Honours (Computer Science)

Faculty of Communications, Health and Science

Edith Cowan University

Date of submission – 9 July 1999
Abstract

Today more IT professionals are employed on the maintenance of existing software applications than are employed to develop new systems. Why is there such a need for this maintenance? Part of the problem is that developers have traditionally seen system requirements as fixed from the time they have been ‘signed off’. In reality requirements are dynamic and subject to change as an organisation’s environment changes.

Flexible software techniques recognise that software requirements are subject to future changes. Flexibility is seen as an important design goal criterion with “true” or “strong” flexibility implying that an application’s behaviour can be altered without the need for changing program code.

The purpose of this study is to:

- Identify flexible software techniques described in the current literature.
- Identify features present in the Oracle suite of tools that can lead to flexibility.
- Design and implement a demonstration application that demonstrates both the flexible techniques and features identified.
Declaration

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

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

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

iii. contain any defamatory material.

Signed: [Signature]

Stephen O’Connor

Date: 25 Mar 2000
Acknowledgements

I would like to take this opportunity to acknowledge and thank some of the people who helped and assisted me in this work.

Special thanks must go to Ms Jean Hall, my supervisor, for the many hours of support, advice and encouragement she gave me during the research and preparation of this paper.

I would also like to thank Mr Roger Edland, a work colleague, who has given me valuable technical advice with the Oracle Development tools.

Finally, thanks to my family for all their support and encouragement during the past two years.
# Table of Contents

Use of Theses..................................................................................................................

Abstract............................................................................................................................

Declaration..................................................................................................................... ii

Acknowledgements....................................................................................................... iii

Table of Contents .......................................................................................................... iv

Table of Figures .............................................................................................................. vi

Glossary of Oracle Developer 2000 terms ................................................................... vii

Chapter 1: Introduction.................................................................................................. 1

1.1. Introduction ............................................................................................................... 1

1.2. Background to the Study ........................................................................................ 1

1.3. What is Software Maintenance? ............................................................................ 3

1.4. The Hidden Cost of Software Maintenance ....................................................... 6

1.5. What is Flexible Software? .................................................................................... 7

1.6. Significance of the Study ...................................................................................... 8

1.7. Purpose of the Study ............................................................................................. 10

1.8. Research Questions ............................................................................................... 11

1.9. Conclusion ............................................................................................................ 12

Chapter 2: Review of the literature ............................................................................. 13

2.1. Introduction ............................................................................................................ 13

2.2. General Literature ................................................................................................ 13

2.2.1. Component based ........................................................................................... 13

2.2.2. Fragment Based Specification ........................................................................ 14

2.3. Literature on Previous Findings ......................................................................... 14

2.3.1. Dynamic Search Condition ............................................................................ 14

2.4. Specific studies similar to the Current Study ....................................................... 15

2.4.1. Common Code Tables ..................................................................................... 15

2.5. User Extensibility ................................................................................................ 16

2.5.1. Data-Driven Navigation Bar ........................................................................... 17

2.5.2. Dynamic Court Orders ................................................................................... 18

2.6. Native Oracle Features that can lead to Flexibility ............................................. 24

2.6.1. Introduction .................................................................................................... 24

2.6.2. %ROWTYPE .................................................................................................. 24

2.6.3. Dynamic PL/SQL and SQL .......................................................................... 28

2.6.4. Dynamic Properties ...................................................................................... 28

2.6.5. Database Triggers ......................................................................................... 29

2.6.6. Database Triggers .......................................................................................... 29

2.6.7. List of Values ................................................................................................. 29

2.6.8. Oracle Roles .................................................................................................. 30

2.6.9. Variable Cursors ........................................................................................... 30

2.7. Conclusion ........................................................................................................... 31

Chapter 3: Method ......................................................................................................... 32

3.1. Introduction ........................................................................................................... 32

3.2. Research Questions ............................................................................................... 32

3.3. Design ................................................................................................................... 32

3.4. Environment ......................................................................................................... 33

3.4.1. Oracle ............................................................................................................ 33

3.5. Demonstration Application ................................................................................... 34

Chapter 4: The Demonstration Application................................................................... 35
# Table of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Approximate costs of software process phases. Schach (cited in Sallis, Tate and MacDonell, 1995, p. 3)</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>Distribution of maintenance activities (based on a study of 487 software development organisations)</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>E-R Diagram for Dynamic Button Bar</td>
<td>17</td>
</tr>
<tr>
<td>4</td>
<td>Initial display of clauses</td>
<td>21</td>
</tr>
<tr>
<td>5</td>
<td>Order screen after text substitution</td>
<td>22</td>
</tr>
<tr>
<td>6</td>
<td>Application E-R Diagram</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>Developer 2000 Object Navigator</td>
<td>37</td>
</tr>
<tr>
<td>8</td>
<td>Canvas layout for the Button Bar</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>User Created System Parameters Form</td>
<td>40</td>
</tr>
<tr>
<td>10</td>
<td>Form Maintenance</td>
<td>41</td>
</tr>
<tr>
<td>11</td>
<td>Item Maintenance Form</td>
<td>41</td>
</tr>
<tr>
<td>12</td>
<td>Form/Item Maintenance Form</td>
<td>43</td>
</tr>
<tr>
<td>13</td>
<td>New Form layout</td>
<td>44</td>
</tr>
<tr>
<td>14</td>
<td>Customer Maintenance Form</td>
<td>45</td>
</tr>
<tr>
<td>15</td>
<td>Screen for company customers</td>
<td>46</td>
</tr>
<tr>
<td>16</td>
<td>Rule Precedence</td>
<td>49</td>
</tr>
<tr>
<td>17</td>
<td>Rule Maintenance</td>
<td>50</td>
</tr>
<tr>
<td>18</td>
<td>Order entry screen</td>
<td>51</td>
</tr>
<tr>
<td>19</td>
<td>Demonstration application database trigger</td>
<td>52</td>
</tr>
</tbody>
</table>
Glossary of Oracle Developer 2000 terms

Block: A block is a collection of interface items such as text items, radio groups, buttons etc that allow users to view and modify their data. There are two different types of blocks:

Base table blocks — usually correspond to columns in data base tables. Typically a collection of items in a block will represent a database table or view.

Control blocks - usually correspond to where buttons are placed on blocks and derived or computed values not based on database columns or tables.

Commit: Permanently saves to the database all unsaved database transactions resident in the buffer/cache.

Cursor: A cursor is a work area in memory where the current SQL statement is stored. For a query, it stores not only the SQL statement but also column headings and the first, or current, row that has been retrieved by the SELECT statement.

Form: A form is a collection of objects that a user interacts with to view and modify database tables. It is made up of windows, canvases, text items, buttons etc. Typically a form will contain a number of different, but related, blocks.

Items: An item usually corresponds to a single data element or field. Similar to a block, it may relate to a database column or can be used as “containers” for generic control information such as summary columns. An item can belong to one and only one block.
LOV: A list of values (LOV) is a modal pick list and visual presentation of data contained in a record group. From this list users can select a single valid value which is normally used to populate an item.

SQL: Structured Query Language is the standard language for interacting with relational database management systems. This standard was approved jointly by the American National Standards Institute (ANSI) and the International Standards Organisation (ISO) in 1992. This was “a revised and greatly expanded standard of SQL under the name International Standard ISO.IEC 9075:1992, Database Language SQL” (Lulushi, 1999, p. 227). SQL is non-procedural in structure and allows users to specify what is to be done as opposed to how to do it, i.e. non-procedural.

PL/SQL: Procedural Language/Structured Query Language. PL/SQL is the procedural language used by Oracle Corporation in its products. It is a subset of ADA providing constructs within it similar to those found in many 3GLs. It provides a flexible way to extend SQL in manipulating database information.

Objects: Oracle Corporation’s release version 8 supports Object-Relational technology with the ability to support object types and collections. They have for some time classified most things as objects in the development environment. Objects can be buttons, windows, canvases, text items etc.

Object Groups: An object group is used to package, or aggregate, several logically related objects into one group. Object groups can be created in an Oracle Developer/2000 Form and can then be copied or sub-classed to other Forms or
applications. An object group is Oracle’s method of facilitating reusable objects.

**Packages:** A package is a PL/SQL construct that is used to group logically related types, procedures and functions. Packages may be defined in Forms, libraries and menus. Packages can be stored on the client application or stored on the server database. Packages stored on the application usually interact with Oracle Forms functionality such as sizing windows, item navigation and displaying error messages amongst other things. Packages on the database are typically used for the retrieval and modification of data from user applications. They are less costly to network resources as one request can be sent to the package and all further processing can take place at the database level.

**Property Classes:** Property classes allow developers to define common attributes and functionality for objects in one place. An object can inherit all the attributes of its property class such as height, width, etc. Changing the definition of a property class changes the definitions of all objects that inherit properties from it.

**Record Groups:** Record groups are structured sets of data used to pass data between the database and application programs most commonly using LOVs. They can be thought of as a virtual table.

**Sequence Numbers:** Sequence numbers are created by developers and are generally used for inserting unique values into primary key fields. Whenever a call is made for a new sequence number, the system automatically increments it by a value specified when the sequence was created.
**Triggers:** Triggers are blocks of code that are used to add functionality to an application. Each trigger contains one or more PL/SQL statements. A trigger can be associated with an event, such as when a new record is created. The code within the trigger executes each time the event occurs.

**Visual Attributes:** Visual attributes are used to set the font name, font size, colour etc of objects. Each object has a property sheet where all the changeable attributes for that object are recorded. The property sheet is the means to set the visual attribute name for a given object. The object will then inherit all the defined properties from that visual attribute.

**Windows/Canvas:** A window by itself is conceptually an empty frame. This frame provides the means to interact with the window including the ability to scroll, move and re-size the window. The contents of the window, or what is displayed inside the frame, is determined by the canvas-view(s) displayed in the window at run-time. A canvas is the structure where objects such as text items, buttons, radio groups etc are laid out. Each canvas must be assigned to a specific window.
Chapter 1: Introduction

1.1. Introduction

This chapter outlines the background to the study, describing software maintenance and the reasons why it has become an important software engineering discipline. Flexible software is defined showing why it is an important design goal. Finally the research questions for this thesis are presented.

1.2. Background to the Study

Today many organisations face unprecedented competition where rapid changes to the way they do business are the norm if they are to remain competitive. They are increasingly turning to the use of Information Technology to gain a competitive edge over their competitors (Callon, 1996, p. 106).

In turn, the costs involved in providing these IT services are under close scrutiny as resources become scarcer and more expensive, financial accountability increases and the global trend towards economic rationalisation continues (Hall & Ligezinski, 1997c, p. 1).

It is widely accepted that the maintenance of software systems can be “the most costly phase of the software life cycle” (Pressman, 1992, p. 667). For the past two decades the cost of this maintenance has increased steadily. The cost of maintenance in the 1970’s was estimated to be between 35 and 40 percent of the software budget, rising to
60 percent in the 1980’s. Pressman suggested that if current trends in software maintenance continued to be followed this cost was expected to rise to 80 percent of an organisation’s software budget during the 1990’s (Pressman, 1992).

Many practitioners acknowledge the high cost of software maintenance in the software engineering community. It is widely accepted that this accounts for anywhere between 60 to 80 percent of an organisation’s IT budget (Sallis, Tate & MacDonell, 1995, p. 2). Sallis, et al, qualify this by saying that although much effort is spent on fixing existing systems, a fair amount of effort is expended evolving existing systems rather than developing new ones.

![Figure 1 - Approximate costs of software process phases. Schach (cited in Sallis, Tate and MacDonell, 1995, p. 3)](image)

A study at the University of California found that “for every dollar spent on application development, more than 50 cents was spent on maintenance” (Asbrand, June 1997, p. 1).
Hewlett-Packard reported that 60 to 80 percent of research and development personnel were involved in maintenance activities involving their 50 to 60 million lines of code. (Brown, Carney & Clements, 1995).

What is happening regarding software maintenance in Australia today? An example from the local Western Australia industry shows considerable effort is being expended on software maintenance. The distribution of IT personnel within the Courts Team, Ministry of Justice (MoJ), Western Australia was given as:

- One third of staff on maintenance tasks.
- One third of staff on development tasks.
- One third of staff undertaking a variety of maintenance and development tasks.

This figure does not include members of the year 2000 team. (C. Blake, Client Manager (MoJ), personal communication, June 14, 1999).

1.3. What is Software Maintenance?

Sallis, et al, have previously stated that not all software maintenance is spent on fixing existing systems. There are other aspects that different practitioners also call maintenance. As with many fields of software engineering, there are many definitions and many arguments about exactly what constitutes a particular discipline. The majority of definitions in the literature and various texts define software maintenance as four distinct activities. (Pressman, 1992, Smith & Votta, 1998, Behforooz & Hudson, 1996)
These activities are defined as:

- **Corrective Maintenance**
  
  Corrective maintenance is the identification and correction of software errors also known as ‘bug fixing’. Humphrey (1997, p.159) estimates that the cost of correcting software errors increases by about ten times in each stage of the development process. He cites the example of IBM who spent about US$250 million “repairing and re-installing fixes to 13,000 customer-reported defects” at a cost of nearly US$20,000 each.

- **Adaptive Maintenance**
  
  Adaptive maintenance modifies existing software so that it conforms to an organisation’s changing requirements.

- **Perfective Maintenance**
  
  Perfective maintenance adds new capabilities, modifies existing functions and makes general enhancements. This accounts for the majority of all effort expended on maintenance.

- **Preventive Maintenance**
  
  Preventative maintenance changes software to improve its future maintainability or reliability or to provide a better basis for future enhancements. Stacey (1995, p.1) states that this type of maintenance which makes software flexible is “still relatively rare”.

4
Preventive maintenance is seen by Kleist (1994, p.20) as being difficult because organisations cannot control all the factors “that can cause the appreciation or depreciation of an information system”.

![Pie chart showing distribution of maintenance activities](image)

Figure 2 - Distribution of maintenance activities (based on a study of 487 software development organisations) Stacey (1995, p.1).

Bartol, Martin, Tein and Mathews (1995, p.82) outline an organisation’s ‘mega-environment’ which they describe as external factors to any organisation that the organisation cannot control. They go on to say that although organisations cannot control them, at least in the short term, they “must be alert for changes in them”.

Bartol, et al, describe the five mega-environment elements as:

- **Technological**
- **International**
- **Sociocultural**
- **Legal-political**
- **Economic**
Kleist (1994, p.19) argues that business change and technology change are the two most significant factors facing IT departments as they “are well beyond the capacity of any single organisation to control”.

1.4. The Hidden Cost of Software Maintenance.

As previously stated, software organisations can spend anywhere from 60 to 80 percent of all funds conducting software maintenance tasks. This is the visible cost to an organisation, however, according to Stacey (1995, p. 1) the hidden costs of maintenance can be even greater because:

- Maintenance-bound organisations result in loss or postponement of development opportunities.
- Customer dissatisfaction when requests cannot be addressed.
- Reduction in overall software quality as a result of changes that introduce latent errors in the maintained software.

Woolfolk, Ligezinski and Johnson (1996, p. 482) give the example of an American factory where new management decided to “flatten” the organisation by removing middle management, combining certain departments and splitting others. They state that although this was not a simple change, it was not uncommon in an organisation’s life. Within six to eight weeks most of those personnel affected had begun using the new procedures and had altered their communication lines etc. Over a year later the factories “computer systems were only 90 percent complete at a cost exceeding a quarter of a million dollars”.

6
1.5. What is Flexible Software?

Flexible software can be seen as the middle ground between professional IT staff maintained systems and end user development. Typically software systems are designed and implemented by IT professionals and at some stage accepted by an organisation and go into production. At this point the maintenance begins. This is usually performed by the IT staff. On the other hand "end user development proposals see end users as developers who take full responsibility for creating their application systems" (Mehandjiev & Bottaci, 1998, p.3). End user development has been criticised because it produces error prone software where development "methods were largely informal" and "fell far short of disciplines long known to be necessary in programming" (Panko, 1998, p. 16).

Flexible software attempts to take the middle ground by having professional IT staff design and implement systems in such a way that the end users can maintain them. Mehandjiev and Bottaci (1996, p. 432) state that organisations that rapidly adapt to changing conditions require flexible software systems as "conventional system development methods are too slow". They see flexible software systems as a method to alleviate the problems caused by conventional systems where end users, or domain experts, can control and modify the behaviour of systems. An alternative name for flexible software is “user enhancability”.

Flexible software techniques recognise that the requirement for systems are, by their very nature, dynamic and attempts to build flexibility into the design, thus reducing the
need for maintenance. Parnas (1979, p.136) describes software as flexible “if it is easily changed to be used in a variety of situations”.

Woolfolk, Ligezinski and Johnson (1996, p. 486) propose a classification of the degree to which software is flexible. They consider the problem of implementing changes in requirements i.e. user enhancability or adaptive maintenance.

- Weak flexibility - if information structure modification is required, e.g. entities or tables.
- Medium flexibility - if only procedural code and data value modifications are required.
- Strong flexibility - if only data values modifications are required.

“‘Strong’ or ‘true’ software flexibility infers that the behaviour of an application can be modified without changing program code” (Hall & Ligezinski, 1997a, p.3).

1.6. Significance of the Study

Blum (1993a, p.43) categorises requirements as ‘closed’, ‘abstract’ or ‘open’. Closed requirements are well defined and stable. He states that there normally exists a “domain notation” that is used to specify these requirements. Examples of closed requirements given by Blum are mathematical notations used in engineering applications.

Requirements are abstract if they have no concrete representation. Blum states that abstract requirements such as software security and safety have no “external reality” and that they “must be modelled abstractly so that analysts can reason about them”. Blum
Requirements are open if the problem domain is poorly understood and/or dynamic. It is this area that flexible software techniques address. Blum states that most applications do no fall into one category but can be characterised by all three.

Many systems today have been designed to implement closed or static requirements. At some point there has been a traditional ‘sign off’ and the system developed around these requirements which were thought to be complete at that time. Behforooz and Hudson (1996, p. 396) argue that “maintainability should be specified and software should provide for the highest level of flexibility and ease of maintenance”. They go on to state that these should be major design goals because:

- The maintenance of software is extremely expensive.
- A system can spend up to 65 percent of its operational life in maintenance.
- The advantages of designing for ease of maintenance far outweigh the costs of including maintenance in the first instance as a major design goal.

As organisations are constantly changing, and changing increasingly rapidly, it is difficult to fully understand the requirements of any system before it is built. The knowledge of these systems is therefore “imperfect since a significant part of such requirements lie in the future” (Woolfolk, Ligezinski & Johnson, 1996, p.482).

Because the environment changes, the initial assumptions can become invalid. As it takes time to identify these changes and implement them in code, the system can quickly become inadequate and “lead to systems that are judged unsatisfactory or unacceptable by the client and have high maintenance costs” (Hofmann, Pfeifer & Vinkhuyzen, 1996, p. 1).
Research into software maintenance has traditionally been a neglected area of the system development life cycle especially when compared to the other phases, however this is starting to change prompted in part by ever increasing maintenance costs (Pressman, 1992).

Software maintenance is now gaining more attention as a software engineering discipline however “the approaches/tools of maintenance are rather weak when contrasted to those of development” (Liu & Zedan, 1998, p. 1). The main reasons they cite are that research into, and the software discipline of development, is mature, while maintenance is seen as difficult and expensive.

1.7. Purpose of the Study

While there is plenty of well known literature on the development of software systems, little seems to have been published on methods for the development of systems where one of the design goals is flexibility. The purpose of this research is to understand the different techniques described in the existing literature that can lead to the development of software systems that exhibit flexibility.

Software systems that exhibit flexibility are not new and applications have been developed for some time in the banking and finance sectors where business rules change rapidly. New, or changing, business rules need to be quickly and easily reflected in these software systems. Currently, the development of flexible software systems has
been confined mainly to 3GL environments. Little has been formalised in a 4GL environment although some 4GL programmers do use available flexible techniques.

One of the goals of this research is to investigate features in a 4GL environment that can be used to enhance the flexibility of software systems. The Oracle suite of tools was used as the preferred 4GL-development environment for the following reasons:

- It is readily available at Edith Cowan University.
- It is widely used throughout the world by medium to large sized Organisations.
- Although the author uses Oracle at his place of employment he had no exposure to Oracle 8, the environment used at ECU.

A sample application was developed using Oracle and the Developer 2000 4GL tools to demonstrate some of the techniques that can lead to flexible applications.

1.8. Research Questions

This study is guided by and should answer the following questions:

1. What is flexible software?
2. What techniques are available that can lead to the development of flexible software applications?
3. What facilities exist in Oracle tools that exhibit flexibility?
4. Can some of the techniques identified in point 2 be implemented using the facilities identified in point 3?
1.9. Conclusion

Chapter 1 described the growing problem that the software engineering community continues to suffer from. It highlighted the problems where systems are designed around closed requirements and suggested that using flexible software techniques is a viable method to reduce the maintenance problem. Chapter 2 describes methods from the literature, which are aimed at improving the software development process and reducing maintenance.
Chapter 2: Review of the literature

2.1. Introduction

Chapter 1 introduced the reader to the problem of software maintenance and suggested that using flexible software was one method to help alleviate this problem. Chapter 2 presents techniques sourced from academic and the software industry that could be seen as flexible and help reduce software maintenance. Finally native features in the Oracle Developer 2000 that can lead to flexibility are described and in most cases examples are presented.

2.2. General Literature

2.2.1. Component based

Parnas (1979, p. 129) argues that a level of flexibility can be achieved by designing software that is easily extended or contracted. His methodology is to identify the minimal subsets that might perform a useful service and then search for the set of minimal increments to the system. For systems to be easily extended or contracted Parnas recommends that:

- Components within the systems should perform no more than one function.
- Each component should not assume that a given feature is present in the system.
- Components should not rely on the output and format of data from another component.
2.2.2. Fragment Based Specification

Blum (1993b, p. 728) describes a representation scheme for software systems where the requirements are not well understood or dynamic. His fragment-based specification is used to capture a conceptual model of the system to be developed. Blum outlines a development environment called TEDIUM. Concepts known about the application to be developed are stored in an Application Database (ADB) as fragments and integrated before a program is generated. This type of facility is available today in 4GLs however Blum characterises them as inefficient.

2.3. Literature on Previous Findings

2.3.1. Dynamic Search Condition

Woolfolk, Ligezinski and Johnson (1996, p. 3) outline flexible software techniques that have been used in banking and finance systems. Their dynamic search condition is used to represent the requirements of the business rules associated with sales commissions. This involves the use of two files and a search program. The first file contains the values of commissions and its determining factors. The second file describes the key that is needed to select the appropriate sales commission from the first file. The program, which is a callable algorithm, accepts input as arguments and searches the first file in the sequence determined by and using the key built by the second file. This means that program modifications are not necessary to accommodate new business rules because new rows, or rules, can be added dynamically (Woolfolk, Ligezinski & Johnson, 1996).
2.4. Specific studies similar to the Current Study

2.4.1. Common Code Tables

One flexible software technique that can easily be implemented in the Oracle Developer 2000 environment is the use of "common code" tables. Code values that are likely to change are not hard coded but rather stored in database tables. When a new code value needs to be added, or an existing one needs modification, they can be accessed at run time through database queries, dynamic record groups and lists of values as opposed to the usual predefined pop-lists (Hall & Ligezinski, 1997c, p. 7). A first level of flexibility is achieved through these dynamic codes. A second level of code flexibility has been demonstrated in an Edith Cowan University project where "...codes can be ‘fuzzy’ e.g.

**Classes** of accounts

**types** of currencies

**groups** of products

**families** of medical risk factors" (Hall & Ligezinski, 1997c, p.7).

It is not necessary to specify the content of classes, types, groups and families at development time as they can be defined by further code values at run time. The Edith Cowan project has implemented second level flexible codes for a system that tracks the outcomes of cancer care developed for the Health Department of Western Australia.
2.5. User Extensibility

Ensor and Stevenson (1997, p.503) discuss user extensibility. They refer to two types of flexibility to categorise applications when requirements are not well understood. These are “schema extensibility” and “algorithmic” extensibility.

Schema extensibility refers to situations when new attributes or entities are required.
Algorithmic extensibility applies when new business rules are needed to supplement, or replace, the current rules. They outline two categories of algorithmic extensibility:

- Data driven extensibility.
- Procedure or function driven extensibility.

Ensor and Stevenson (1997, p.504) state that if “a rule, rather than the value of a term within a rule, is subject to change, then you may be looking at a requirement for extensibility”. Ensor and Stevenson suggest that when a rule is likely to change, the code used to implement the rule should be stored as a package in a data base table. This allows for the addition of new, and modification of existing, rules.

Procedure and function driven extensibility can be achieved by ensuring that every atomic action taken by an application is stored in the data dictionary as procedures or functions and used as required. This means that the code is only stored once but maybe used, or called, many times, leading to applications that are easier to maintain. The down side to this is the overhead of having many small functions or procedures.
2.5.1. Data-Driven Navigation Bar

Membrey (1999, n.d, p. 1) describes a flexible data-driven dynamic navigation bar. He outlines the concept by placing ten buttons on an Oracle Developer 2000 Forms canvas. These buttons are based on a property class, which includes a dummy WHEN-BUTTON_PRESSED trigger. This trigger is used to call a yet to be nominated Form.

Forms object types are stored in a database table called APP_OBJECT and can include windows, forms, text items, buttons etc. A second table called APP_OBJECT_NAVIGATE is used to store which particular Form an object button should call when clicked.

![E-R Diagram for Dynamic Button Bar](image)

**Figure 3 - E-R Diagram for Dynamic Button Bar**

Two foreign key relationships exist from app_object_navigate:

Object_name, object_type → app_object.object_name, app_object.object_type

Object_target → app_object.object_label

When a button is pressed, the dummy trigger ‘fires’ and dynamically searches the database to determine which Form it should call. Whenever a Form is opened it first searches the table APP_OBJECT_NAVIGATION and retrieve all the possible buttons associated with that Form. It then dynamically displays them one beneath the other.
2.5.2. Dynamic Court Orders

The following sections, regarding Court Orders, have been included with the permission of Eileen Magyar, Project Manager, Courts, Ministry of Justice, Western Australia.

Various levels of flexibility have been demonstrated in a Ministry of Justice IT project by the author. This involved the design and implementation of court orders for the Guardian and Administration Board, part of the higher courts in Western Australia.

Traditionally, standard headings, titles and text etc had been hard-coded into reports. This necessitated IT resources whenever requirements, such as changes to the legislation, were made.

2.5.2.1. Design

The different sections of the court orders were grouped into functionally similar types. These types were then stored on the database. Headings, address blocks and the different paragraphs, known as “Clauses”, were assigned to a type and again stored on the database. The final design of the database was a three tiered system whereby the clause’s and information about those clauses were stored in a number of different tables.

- 7 tables were used to store the clause text and information about them.
- 3 tables described the orders, with references to the database tables outlined in point one. Through these tables, each order “pointed” to the different component clauses that were required, at run time, to build the order.
- 3 tables were used to store the final “built” orders.
At run time users select the type of order they want to make from a list of values. The order is then dynamically built based on the information about that order stored in the previously described tables. The clauses within these orders are also dynamically built based on a callable algorithm. This algorithm, shown below, was used to build all the clauses when the order was first selected. It was also used to substitute variable text into individual clauses when users selected them for inclusion on the order.

While there are clauses for this order
LOOP
  While there are rows for this clause
  LOOP
    SELECT the text and lookup value
    LOOP
      IF the lookup value IS NOT NULL THEN
        -- Determine the substitute value --
        IF the lookup value is a string enclosed by #[]#' and the clause has been selected THEN
          display a text editor to the screen for user input.
        ELSE
          Insert the string enclosed by #[]#' into the clause as text
        END IF
      ELSE
        IF lookup value = 'CHR(10)' and the order has just been selected THEN
          -- Carriage return character, force a carriage return into the text --
        ELSE
          Lookup value refers to a screen text item
          Get the value in this field and do a lookup in the substitute table
          IF a match is found THEN
            get the text and insert it into the clause
          ELSE
            insert the value found in the text item into the clause.
          END IF
        END IF
      END IF
    END LOOP
  END LOOP
END LOOP
END
Table 1 below shows how one clause is stored within the database. Clause No, a foreign key, references the Clause table which stores what type of clause it is and other information such as date created and date retired. A clause is built at run time by concatenating its various component parts together. Any clause text that is enclosed by the character string of #[]# is variable text and will be substituted at run-time by a number of yet to be determined factors.

<table>
<thead>
<tr>
<th>Clause No</th>
<th>Row No</th>
<th>Clause Text</th>
<th>Lookup Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>is unable, by</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>#mental disorder, intellectual handicap or other mental disability#</td>
<td>24</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
<td>to make reasonable judgements in respect of matters relating to all or any part of</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>#His/Her#</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>Estate</td>
<td></td>
</tr>
</tbody>
</table>

Table 1 – Sample Clause
Figure 4 below shows how the clause shown in table 1 and some of the other clauses are displayed when the order is built and displayed to the screen.

When the order is first built it displays all the standard clauses for that order. The user then have the choice of which clauses they want in the court order depending on hearing outcomes. They also have the ability to add new, ad hoc clauses if required. The ability to change the order in which the clauses appear on the report is provided by a sequencing field on the Form.

Not all variable text can be determined dynamically. In the example shown in Table 1 the users could select any combination of the variable text enclosed in the #][# string. When the check box next to the clause was selected a text editor containing the text string was presented for editing. When the text editor was dismissed the new text was
inserted into the clause. Figure 5 below shows how the screen may appear after the sample clause has been edited.

![Order screen after text substitution](image)

**Figure 5 - Order screen after text substitution**

Variable text within clause’s are determined and substituted at run time based on not only the look up value but also a number of other criteria. Examples of which are:

- Gender of the party to whom the order pertains.

  In the example shown above the clause row number 4 #[His/Her]# will be replaced with “His” or “Her” depending on the gender of the person to whom the clause refers to. These values are stored on the database and retrieved dynamically.

- Plurality of text. For example if there is only one applicant for the order then “Administrator sees fit” is displayed or if there are more than one applicant...
“Administrators see fit” is displayed. Again the number of applicants is determined dynamically at run time.

Further data entry screens enable certain end-users, known as ‘domain experts’, to modify existing orders and clause’s and also to add new ones.
2.6. Native Oracle Features that can lead to Flexibility

2.6.1. Introduction

The Oracle development environment contains many features that can be used for flexible application design. Various small prototyping exercises were carried out to determine the suitability of certain features for use in the demonstration application. These include:

2.6.2. %ROWTYPE

The identifier %ROWTYPE declares a record variable that has the same structure as:

- a row in a table or view or;
- a row retrieved by a cursor.

It is used in the variable declaration section of procedures, functions and packages to ensure that when the variable is assigned it contains the correct fields and data types of the columns being fetched. Consider the following database table ST_FORM and the simple PL/SQL procedure block:

<table>
<thead>
<tr>
<th>Column Names</th>
<th>Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORM_ID</td>
<td>ST_FORM</td>
</tr>
<tr>
<td>FORM_NAME</td>
<td></td>
</tr>
<tr>
<td>FORM_TITLE</td>
<td></td>
</tr>
<tr>
<td>FORM_WIDTH</td>
<td></td>
</tr>
<tr>
<td>FORM_HEIGHT</td>
<td></td>
</tr>
<tr>
<td>FORM_START</td>
<td></td>
</tr>
<tr>
<td>FORM_END</td>
<td></td>
</tr>
</tbody>
</table>
DECLARE
    Form_record st_form%ROWTYPE
BEGIN
    SELECT *
    INTO Form_record
    FROM st_form
    WHERE form_name = 'ITEM' ;
    IF Form_record.width > 1000 THEN
        Do_some_thing
    END IF ;
END ;

The variable Form_record is declared to have the same structure as a row from the
database table st_form. When the variable is assigned it can reference the
FORM_WIDTH field as VARIABLE NAME.COLUMN i.e.
Form_Record.form_width

When a cursor is used as the prefix for %ROWTYPE then restrictions can be placed
on the columns that will be returned. In the following example only the fields height
and width are returned.

DECLARE
    CURSOR cur_form IS
    SELECT height, width
    FROM st_form;
BEGIN
    OPEN cur_form
    FETCH cur_form INTO Form_record
    IF Form_record.width > 1000 THEN
        Do_some_thing ;
    END IF ;
END ;

Similarly, %TYPE is used to declare a new variable to be of the same type as a
previously declared variable, or a column in a table, that exists in the database.
The real value of using \%ROWTYPE and \%TYPE is only exploited to the full when used on database procedures, packages etc. The problem lies in how client and server applications are compiled. Consider the following function from the demonstration application:

```sql
FUNCTION get_window_width (p_window_name VARCHAR2) RETURN NUMBER IS
  -- Declaration section --
  CURSOR cur_width IS
    SELECT form_width
    FROM st_form
    WHERE form_name = p_window_name;
  lv_width st_form.form_width%TYPE;
BEGIN
  OPEN cur_width;
  FETCH cur_width INTO lv_width;
  CLOSE cur_width;
  RETURN (lv_width);
END get_window_width;
```

In the declaration section the following variable was declared

```
lv_width st_form.form_width%TYPE;
```

This declares the variable `lv_width` to be the same type as the column `form_width` in the table `st_form`. If the function `Get_Window_Width` is stored as a database procedure and the type of the column `form_width` is changed (from numeric to character) then it will be marked as ‘invalid’. The next time the procedure is called it will be compiled automatically before it is run. If, on the other hand, the procedure is stored on a client side application such as Oracle Forms, the next call to the procedure will raise an exception. This is because the Forms source is compiled at a fixed point in time and the executable will still try to reference the table using the old data type. This demonstrates the added flexibility achieved by storing as much code as is practicable on
the database. If the code was stored on the client, changing one column type could necessitate the re-compilation of an entire application.

According to Stacey (1995, p. 1) changes in data formats account for the greatest share of software maintenance activities. Stacey sees data structure changes as the main problem because the effect is very rarely localised. The knowledge, or use, of a particular data structure is often spread over many parts of any system therefore making the change costly.

Feuerstein (1996, p. 25) states that one of the most common causes of application failure is the “undying belief held by programmers that a particular value will never change and so can be hard coded into the program”. He goes on to say that requirements change on a daily “if not hourly basis”. Feuerstein advocates the storage of constants in a PL/SQL package specification. If the value of a constant changes then the package specification can be updated with change localised to one place. The only problem with this method is that all the packages dependant programs must be re-compiled.

The problem of database columns changing type was seen by the author in a Ministry of Justice project that upgraded the Children and Petty Sessions Case Management System. In the old system, charge numbers were stored as characters and on the new, converted system, as numbers. If all the variables used to store charge numbers had been declared as characters, it would have resulted in a great deal of effort to change them by the developers. As %ROWTYPE and %TYPE had been used then the code had only to be re-compiled.
2.6.3. Dynamic PL/SQL and SQL

One of the limitations of PL/SQL is that it cannot contain Data Definition Language (DDL) statements such as creating and dropping tables. The Oracle DBMS_SQL package provides the facility to insert these SQL statements into PL/SQL at run time. The statements are put into a string, parsed and executed dynamically.

The DBMS_SQL package offers access to dynamic SQL from within PL/SQL. Dynamic SQL statements are not pre-written into programs. They are constructed at run time as character strings and then passed to the SQL engine for execution.

2.6.4. Dynamic Properties

Properties of object types may be modified and set at run time via the GET and SET property functions. Examples of these are:

- Maximise the MDI window
  
  Set_Window_Property(FORMS_MDI_WINDOW, WINDOW_STATE, MAXIMIZE);

- Get the width of the window named WIN_MAIN
  
  Get_Window_Property('WIN_MAIN', WIDTH);

- Do not allow users to enter new records into the block called st_customer
  
  Set_Block_Property('st_customer', INSERT_ALLOWED, PROPERTY_FALSE);

- Display the item named nav_itms.item_name
  
  SET_ITEM_PROPERTY (nav_itms.item_name, VISIBLE, PROPERTY_TRUE);
2.6.5. Database Triggers

Both triggers and PL/SQL code can be stored in the database on the server, as opposed to on the application client. The advantage of this is that code needs only be stored in one place. It is then available to any application that has access to that database. Changes to triggers or procedures used by applications need only be changed in one place. Database triggers attached to tables can be used to enforce business rules at the database level ensuring that whoever accesses that table will follow the rules consistently.

2.6.6. Dynamic Record Groups

Dynamic Record groups can be created and populated by using SQL. They can also be created with the existing facilities in Oracle Forms at run time.

2.6.7. List of Values

Lists of Values can be used in applications instead of traditional pop lists. As they are associated with record groups, this ensures the information they contain is always current. Unlike the traditional ‘pop-lists’ using the function SHOW_LOV (see example below) does not necessitate the LOV to be attached to a text item. For example, you can use SHOW_LOV to allow end users to invoke a LOV by clicking a button or selecting a menu item. The SHOW_LOV function is a BOOLEAN function that returns TRUE if the end user makes a selection from the LOV and FALSE if the end user cancels the LOV without making a selection.
The simplest way to call the SHOW_LOV function is to assign the return value of SHOW_LOV to a dummy variable, as shown in the following When-Button-Pressed trigger.

```sql
DECLARE
  dummy BOOLEAN;
BEGIN
  dummy := Show_LOV('my_lov',15,10);
END;
```

### 2.6.8. Oracle Roles

Security roles control access to menus and application functionality. They can be defined and modified dynamically. An Edith Cowan Honours project has extended this concept whereby user access can be controlled based on time of day, terminal used or other predetermined factors (Layng, 1998).

### 2.6.9. Variable Cursors

Variable cursors can be declared and assigned dynamically at run time. The following simple example shows how a variable cursor is associated with different tables at run time depending on user input.

```sql
IF : user_Variable = 1 THEN
  /* Open variable for Department table */
  OPEN : flexible_Cursor FOR
  SELECT DeptID, DeptName
  FROM Dept;
ELSE
  /* Open variable for Employee table */
  OPEN : flexible_Cursor FOR
  SELECT EmpID, EmpName
  FROM EMP;
END IF;
```
2.7. Conclusion

This chapter described some of the techniques that have been described in the literature as methods that can lead to the development of flexible software. Also presented were techniques that have been used in live systems and the reasons that they have reduced software maintenance. Features and examples from the Oracle Developer 2000 environment that can be used to develop flexible software techniques were also described. Chapter 3 discusses the methodology used to answer the research questions presented in chapter 1 and presents the techniques and Oracle Developer 2000 features that were used in the development of the demonstration application.
Chapter 3: Method

3.1. Introduction

This chapter outlines the methodology undertaken to answer the research questions proposed in chapter 1 and reproduced below. Chapter 3 also describes the Oracle environment used to develop the demonstration application. The final section of the chapter outlines the flexible software techniques that were implemented using the native Oracle features described in chapter 2.

3.2. Research Questions

1. What is flexible software?
2. What techniques are available that can lead to the development of flexible software applications?
3. What facilities exist in Oracle tools that exhibit flexibility?
4. Can the techniques identified in point 2 be implemented using the facilities identified in point 3?

3.3. Design

Research questions 1 and 2 were answered by canvassing a number of sources, which included:

- Papers published in the academic literature.
- Current literature on the Oracle suite of development tools sourced from industry.
- Information obtained from various Oracle User Groups and Oracle related Web sites.
- Information from current practitioners who currently user the Oracle suite of tools.
- Implementing a number of small prototype applications, in the Oracle Developer 2000 suite of tools, that demonstrated flexibility.

Research question number 3 was answered reviewing the available features of Oracle 8 and building a number of small prototypes.

Research question 4 was answered by designing and implementing a small Oracle application that demonstrated flexible software techniques using the Oracle 4GL environment. This was by far the most time consuming part of the project. The application designed was based on the standard Customer/Order/Product set of database tables that is shipped with Oracle for training purposes. The application was never intended to meet the full requirements of a customer order system but merely as a method to demonstrate various techniques for implementing flexibility.

3.4. Environment

3.4.1. Oracle

Oracle Enterprise Edition version 8 on a Windows NT PC using a local database and single Repository was used for the development of the demonstration application. Specifically, the demonstration application was built using Forms Developer/2000 (32 bit) Version 5.0.6.8.0.

Developer 2000 and specifically Forms Designer were chosen as the preferred development environment for the application front end. Developer 2000 was
considered “the most powerful client/server application development environment for Oracle databases” (Lulushi, 1999, p. 80). Developer 2000 and the Oracle server database share the same language (PL/SQL) and Developer 2000 was designed exclusively to support Oracle database functionality.

3.5. Demonstration Application

The demonstration application was developed to determine if the flexible software techniques from the literature could be implemented using native Oracle Developer features. These techniques and features were discussed in chapter 2.

The concept of components within the system performing no more than one function, outlined by Parnas in chapter 2, was demonstrated by the use of packages. Appendix B shows how this idea was implemented.

User extensibility, as described by Ensor and Stephenson, was demonstrated in the implementation of the dynamic search condition, described by Woolfolk, Ligezinski and Johnson in section 2.3.1. This was one of the main features of the demonstration application. The dynamic search condition is discussed in greater depth in chapter 4.

Features of the demonstration application were presented at Edith Cowan University during a Honours and Masters degree presentation day held at the Mount Lawley Campus on the 16 May 1999.

The results of the implementing flexible software techniques in a 4GL environment will be presented in chapter 5.
Chapter 4: The Demonstration Application

4.1. Introduction

This chapter outlines the flexible software techniques highlighted in chapter 3 that were implemented using the features identified as flexible in the Oracle developer environment.

4.2. Demonstration Application

The standard Customer/Order/Product database tables that ship with Oracle products were used as the starting point for the demonstration application. These tables were extended and modified to incorporate the concepts to be demonstrated. The application was split into two distinct areas. The first area contained the standard functional requirements of the Customer/Order side of the business and its rules. The second part was concerned with the maintenance of application objects, such as the displaying of Forms and items on those forms. Figure 6 below shows the final design of the demonstration application with the tables added to the standard Oracle Customer/Order/Product tables shaded.
Figure 6 - Application E-R Diagram

The Customer/Order/Product tables of the application are composed of relatively standard attributes and will not be expanded on. The entities that maintain application objects are described in table 1 below.

<table>
<thead>
<tr>
<th>Table Name</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST_FORM</td>
<td>Stores the names and characteristics of Forms that are present in the system. Includes the width and height of the Form to be displayed at run-time.</td>
</tr>
<tr>
<td>ST_ITEM_TYPE</td>
<td>Stores the name of all item types that can appear on any given form. These include buttons, text items, LOVs and radio groups</td>
</tr>
<tr>
<td>ST_ITEM</td>
<td>Stores the names of all possible items that can be included on any Form</td>
</tr>
<tr>
<td>ST_FORM_ITEM</td>
<td>Stores the items that will appear on a particular Form and how various different run-time properties will be set when that Form is run.</td>
</tr>
<tr>
<td>ST_FORM_REF</td>
<td>Stores the names of items that should be displayed on a Form given the values stored in a different item. E.g. Different items are displayed on the Customer maintenance form based on what group they belong to.</td>
</tr>
</tbody>
</table>

Table 2 – Table functionality
4.2.1. Templates

Today developers rarely create Forms from scratch but take an existing Form and “delete all base blocks and then customise the Form for the particular application” (Catalano, 1999, p. 1). Templates usually include all the reusable components that will be common to all Forms. The demonstration application extends this concept by placing on the template Form all code and objects that will be used in the application. This then becomes more than just a template but rather an Object library. Related objects such as windows and canvases are placed in Object Groups and can then be subclassed, as opposed to copied, into different application Forms. Changes made to the template Form are automatically propagated throughout the entire application. Figure 7 below shows the design of the template form with various object groups used in the application.
4.2.2. Implementation of the Dynamic Button Bar

The dynamic button bar was implemented in the demonstration application and its functionality was extended beyond the initial idea reported by Membrey (n.d., p. 1). Object names and values for object properties were stored in database tables. This enabled them to be dynamically changed at run time. To limit the scope of the project, only sub-sets of the available object properties were stored. These include the most obvious candidates for change, such as the height and width of Windows and whether items such as text items should be displayed and enabled for user entry. This functionality could easily be extended to cater for a greater variety of object behaviour.

Whenever a new Form is called and displayed, the **WHEN-NEW-FORM-INSTANCE** trigger fires. This trigger is inherited from the template Form. Each Form has a unique name and this is stored in a Form level global variable called: **GLOBAL.gv_form**. This variable is required for navigation between the different Forms. The variable is set with the following assignment statement:

```
:GLOBAL.gv_form := NAME_IN('SYSTEM.CURRENT_FORM') ;
```

The next step is to determine which buttons should be displayed for the called Form and then to display them. The buttons are placed on a separate canvas within its own window at design time. Figure 8 below shows the initial design time layout for the buttons that will be inherited by each Form. A cursor was used to select all the buttons from the table ST_FORM_ITEM for the called Form. Each button was then dynamically placed on the Form. The order of display for the buttons was from left to
right across the screen, dynamically, based on the width of the previous button. When the maximum number of buttons that can be displayed is reached, a new row of buttons is started below the preceding row. The width and height of the window on which the buttons are displayed are dynamically changed each time a new button is added. Finally, the Form window is displayed if there are one or more buttons displayed.

Figure 8 - Canvas layout for the Button Bar

The initial layout has eighteen buttons, three wide and six deep. This is not necessarily how they will be displayed at run-time. As this is a rule that can be subject to change it has been stored on the database in a parameter table. When the call to set up the buttons is executed, the number of buttons to be displayed across the window is determined by the following database call:
lv_mod NUMBER := st_pkg.get_parameter_value ('MAX_ROW_LENGTH');

The database function call `get_parameter_value` passes in the name of the required value as a parameter and the returned value is assigned to the variable `lv_mod`.

Similarly other values such as a windows' width and height are determined by database function calls. End users can maintain the values of these different, developer created, parameters using the form shown in Figure 9 below.

![Figure 9 - User Created System Parameters Form](image)

When a parameter is changed and saved to the database, the changes come into effect the next time a call is made to the relevant function. These changes could easily be reflected dynamically but would be too unsettling to end-users if the layout of screens changed while they were still using them.
4.2.3. Forms Maintenance

Physical details about Forms are stored in the database. Figure 10, from the demonstration application, shows the details that were maintained i.e. width, height and title of the Form. Also a start and end date is stored so that new Forms can be created and old Forms “retired”.

![Figure 10 - Form Maintenance](image)

All the potential items that can be displayed on the application are stored in the database. The Form shown in Figure 11 maintains these items.

![Figure 11 - Item Maintenance Form](image)
Item names are prefixed by the name of the block they belong to. Details such as item type, start and end dates are recorded. If the item is a button then the name of its icon is stored so that it can be associated with the button at run-time.

The ‘Form/Item Maintenance’ Form shown in Figure 12 is used to maintain details of items that will be dynamically set whenever a Form is displayed. It is the means of linking the Forms and Items described above with one another. For each item an arbitrary number of different properties are stored. For the purposes of the demonstration application some of the most commonly used properties are selected. This list is by no means complete and any number of other properties could have been included. The properties chosen were:

- X position
- Y position
- Width
- Height
- Enabled
- Visible
- Label

An extra field is added for special processing. In the case of an item being a button, on the dynamic button bar, the PROMPT field is used to store the name of which Form it should call. In the case of a radio group it is used to store the name of the individual radio buttons.
When the window is first displayed a call is made to the template package to display the appropriate items. Initially all items on the screen are non-visible. A cursor is used to retrieve the names and property values of all items that should be displayed on the called Form. A loop then sets the properties of these items depending on the values retrieved from the database.
Figure 13 below shows the result of changing the properties of the Form shown in Figure 10 through the screen shown in Figure 12.

![Figure 13 - New Form layout](image)

A further level of flexibility is achieved through the use of the ST_REF_ITEM table. According to Hall and Ligezinski (1997b, p. 5) if certain functionality is not available it should be shown “dimmed out” rather than not shown at all. In the case of the demonstration application certain data entry fields pertain only to certain groups of customers. Whenever a user moves to a new record in the customer maintenance screen all items on the Form, excluding the customer ID field, are set to non-visible. A look up is then made to the database to determine which items should and should not be displayed. This is determined by which group a customer belongs to. In the example shown in Figure 14 details such as title, date of birth, given names and surname are displayed if the selected customer belongs to the PERSONAL customer group.
If the customer does not belong to the PERSONAL group then individual fields such as gender, given names and surname have no relevance and are not displayed. Currently there are four customer groups in the system being:

- PERSONAL
- COMPANY
- PARTNERSHIP
- TRUST

Figure 14 - Customer Maintenance Form

Figure 15 below shows the effect of changing from a customer belonging to the PERSONAL group to a customer belonging to the COMPANY group. The fields that pertain to a company replace the irrelevant fields that pertain to an individual. Similarly, different fields are displayed for other customer groups.
4.3. Implementing the Dynamic Condition Search

The concept of the dynamic condition search, as outlined by Woolfolk, Ligezinski and Johnson in section 2.3.1, was implemented in the demonstration application. The contents of File 2 can be seen in Figure 16. The precedence of a given condition over another condition is determined by the index. The index and condition ID are the compound primary key of the rule precedence table. If this were not the case then the table would simply be a table of salesman commissions.
When a new order line was entered on an order the `CALCULATE_COMMISSION` function was called with the following parameters:

- Condition
- Customer Group
- Customer ID
- Salesman Group
- Salesman ID
- Product Group
- Product ID

A "search key" is then built using an SQL cursor to search for the applicable commission rate for the input key. This cursor is used to loop through all the rows that apply to the salesman condition. On each iteration of the loop the values of the fields are examined and if they are equal to one then the value in the search key is replaced with the actual value from the calling program. Once the search key is built the rule maintenance table is searched using a second SQL cursor. This acts as an inner loop to the first cursor. On each iteration of the loop the values in the maintenance table are compared with the search key. If an exact match is found then the condition has been determined and control returns to the calling program. For a full definition of the PL/SQL code used see appendix B.
Consider the following example. The calculate commission function is called with the following parameters.

- **Condition** = ‘COMM’
- **Customer Group** = 2
- **Customer ID** = 1000
- **Salesman Group** = 1
- **Salesman ID** = 2000
- **Product Group** = 1000
- **Product ID** = 3000

On the first iteration of the loop where the index is 1, the search key is built with the actual value of the salesman group of 1 and the rest of the key values as 0. The inner loop searches the table rule maintenance and as no exact match is found repeats the process for index value 2. On the third iteration the search key will be built with a customer ID of 1000, salesman group of 1 and a product group of 1000. All other fields will be set to 0 according to the record in rule precedence.
<table>
<thead>
<tr>
<th>Cond</th>
<th>Index</th>
<th>C Grp</th>
<th>C ID</th>
<th>S Grp</th>
<th>S ID</th>
<th>P Grp</th>
<th>P ID</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMM</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Salesman group overrides all others</td>
</tr>
<tr>
<td>COMM</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Applies to customers only</td>
</tr>
<tr>
<td>COMM</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Products sold to specific customers</td>
</tr>
<tr>
<td>COMM</td>
<td>4</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Products group sold to specific customers</td>
</tr>
<tr>
<td>COMM</td>
<td>5</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>Products sold to specific customer groups</td>
</tr>
<tr>
<td>COMM</td>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>Specific products</td>
</tr>
<tr>
<td>COMM</td>
<td>7</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>salesman salesperson</td>
</tr>
<tr>
<td>COMM</td>
<td>8</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>General rate end of search</td>
</tr>
</tbody>
</table>

Figure 16 - Rule Precedence
As can be seen in Figure 17 a match is found when condition 5 is reached. The value of the commission is retrieved and as a match has occurred this value will be returned to the calling program and the function calculate commission exited as a match with the highest precedence has been found.

Figure 17 - Rule Maintenance
The commission is then calculated in the application Form and the relevant fields updated (See Figure 18 below).

Figure 18 - Order entry screen

To ensure that a match is always found the last record in the rule precedence is all zeros. If this is reached then this will match the first entry in the rule maintenance table. The precedence of these rules can be changed to reflect changing business rules. For example if the values in index 5 and 6 are swapped then the importance of products in product group 1 will have a higher determining factor than products in product group 1 sold to customers in customer group 1.
4.4. **Implementing Database Triggers**

The database trigger shown in Figure 19 is used to insert system generated sequence numbers into new records at the database level. The sequence number is generated by a database package call and a unique number is assigned to the primary key of the customer table. The function that the trigger calls is used to highlight the flexibility of dynamic SQL. This is presented in section 4.4.

```plaintext
4CCONNOR| ST_CUSTOMER| TRIG_CUST_ID

Triggering: Before

Statement: INSERT

For Each Row

Referencing OLD As: OLD     NEW As: NEW

Trigger Body:
BEGIN
    :NEW.CUST_ID := st_comm.Get_Sequence_Number ('ST_CUST_SEQ');
END;
```

**Figure 19 – Demonstration application database trigger**
4.5. Implementing Dynamic SQL

Dynamic SQL is demonstrated in the demonstration application. Whenever a new customer is created, the next customer ID is dynamically retrieved and allocated. When a new record is committed to the database by the application, the trigger body executes. The function `Get_Sequence_Number ('ST_CUST_SEQ')` is called within the trigger body shown in Figure 19. This function is stored within a database package. The function is generic in that it can be used to retrieve any sequence number. Dynamic SQL is the method used to construct and execute SQL statements dynamically. The SQL statement to be executed is “built”, as a character string, using the input parameter. A cursor is opened and the character string parsed to ensure that it conforms to valid SQL syntax. A column is then defined to store the result of the SQL statement. Finally the statement is executed and a sequence number is returned to the calling program.
Chapter 5: Results

5.1. Introduction

Chapter 4 described how flexible software techniques described in the literature were implemented in the demonstration application using native Oracle features. This chapter addresses the results from implementing the demonstration application.

5.2. Research Questions 1 and 2

Research questions 1 and 2 were answered by researching the current literature to find what various people in academia and industry defined as flexible software. Many different definitions were found but most authors recognised that requirements are not fixed in time but are dynamic. A useful definition of flexible software is thought to be the following:

"software flexibility infers that the behaviour of an application can be modified without changing program code" (Hall & Ligezinski, 1997b, p.3).

A number of different techniques that exhibit flexibility were found in the literature. While some were academic in nature the majority of techniques were described by IT practitioners from the software industry.
5.3. Research Questions 3 and 4

Research question 3 and 4 were answered by designing and implementing a demonstration application. The techniques that lead to the development of flexible software applications highlighted in research question 2 were implemented by the Oracle features in research question 3. The techniques were successfully implemented using the Oracle development suite of tools. The following lists includes some of the native Oracle features that were successfully used in the demonstration application:

- List of Values (LOV)
- Dynamic SQL
- Dynamic Record Groups
- Dynamic Properties
- Database Packages
- Database Triggers
- Templates
- Object Groups
- %ROWTYPE and %TYPE

A number of issues arose that did not support the use of some techniques that were thought to lead to flexibility. These issues are discussed below.

5.3.1. Data Driven Button Bar

The data driven button bar was proved to be a practical example of implementing flexible and common functionality in the Oracle 4GL environment. Property classes enabled this common functionality to be placed in one location and then inherited by
various different Forms. The advantage of the button bar is that the functionality is always visible to the user, as opposed to menus, and the user is only one mouse click away from navigation to a different Form. The real issue is that of space the buttons take up on the screen. This was partially overcome by making the button bar ‘floating’ whereby the users could move the bar out of the way if required. Another issue was that of standardisation. Preece (1993, p. 72) suggests that the best way for users to locate a certain piece of information, or functionality, is by using a consistent format for all application screens. The dynamic button bar does not conform to this as the same button may be placed in different locations on different screens.

5.3.2. Dynamic Screen Layout

The dynamic screen concept appeared a good idea in theory however it does have many limitations in practice. User customised screen layouts, outlined by Hall and Ligezinski (1997b, p. 5) and implemented in the demonstration application was found to have severe limitations in the Oracle Developer 2000 environment. One of the main problems was that the ‘rules’ about how forms and items on those Forms should displayed are stored in the database. For small applications with few users this is not a critical issue. However for large applications with many users then efficiency and hits on the database becomes more of an issue as valuable server resources are being used just to implement screen design as opposed to core business functions.

Ensor and Stephenson (1997, p. 507) state that the data-driven approach to application design is not always the best solution for flexibility “as it can lead to something so abstract that it either can’t be coded by mere mortals…or becomes horribly inefficient”.

56
They go on to say that applications where fields are sized and placed dynamically on screens are “desperately slow and impossible to maintain”.

This maintenance problem was illustrated in the demonstration application. Great care had to be taken to ensure that the values stored in the database conformed to Oracle Forms logic. Storing the ENABLED and VISIBLE properties of items highlighted an example of this. These properties can be set to either TRUE or FALSE. The problem with allowing end-users to maintain these values for Form items is that certain combinations are not valid. An example of this is setting an item's ENABLED property to TRUE when its VISIBLE property is set to FALSE. The ENABLED property allows users to navigate to that particular item, i.e. place the cursor in it. This raises an exception, as it makes no sense to allow navigation to an item that is not visible. The demonstration application only used a sample of the available item properties whose values were stored on the database. There are other combinations of property values that cannot be used in conjunction with each other.

The only solution to get around the different combinations is to code the rules, which defeats the purpose of flexible software. It could be argued that the cost of this extra work would be minimal and a ‘once off’ cost once written however these properties can change with different releases of Oracle software. In Oracle Forms Version 4.5 the display of items is controlled by the property DISPLAYED, however in Oracle Forms version 5.0 this has been replaced by the property VISIBLE. This leads to extra maintenance in updating the logic behind the properties.
An even greater problem occurs when an organisation uses different versions of Oracle Forms for different applications. The ideal situation is to store the logic in the database and allow all different applications to access it. If different versions of Oracle Forms are being used by different applications then this will require different versions of the code, which in essence is the same.
Chapter 6: Conclusion

Flexible software techniques have been demonstrated and have proven to be successful in a 4GL environment. These techniques, when appropriately used, can assist in reducing the cost of software system maintenance. The advantages of using these techniques are that they not only reduce maintenance, but increase user satisfaction and ownership of their systems. They have the ability to change system behaviour in a timely manner which in turn can accurately reflect changing business requirements.

The main problem with flexible software techniques is they can lead to systems where response times are compromised in favour of flexibility.

6.1 Recommendations

It is strongly recommended that software developers of 4GL applications be aware of techniques that can lead to flexibility. Developers should be encouraged to use and enhance these techniques where they prove to be cost effective. Developers can learn about flexible software techniques by reading the latest trade journals, attend relevant user groups and keep up to date with the latest trends and techniques.
BIBLIOGRAPHY


Parnas, D.L. (1979). Designing Software for Ease of Extension and
Contraction, IEEE Transactions on Software Engineering, SE-5, (2) 128-137.


# Appendix A – Create Objects Scripts

```
DROP TABLE ST_CUSTOMER CASCADE CONSTRAINTS;

CREATE TABLE ST_CUSTOMER (  
  CUST_ID NUMBER(4) NOT NULL,  
  CUST_GIVEN_NAMES VARCHAR2(40),  
  CUST_SURNAME VARCHAR2(40),  
  CUST_TITLE VARCHAR2(6),  
  CUST_ADDRESS_1 VARCHAR2(40),  
  CUST_ADDRESS_2 VARCHAR2(40),  
  CUST_ADDRESS_3 VARCHAR2(40),  
  CUST_SUBURB VARCHAR2(20),  
  CUST_POSTCODE VARCHAR2(10),  
  CUST_STATE VARCHAR2(4),  
  CUST_COUNTRY VARCHAR2(20),  
  CUST_PHONE NUMBER(14),  
  CUST_FAX NUMBER(14),  
  CUST_CR_RATING VARCHAR2(10),  
  CUST_GRP_ID NUMBER(4),  
  CUST_SEX VARCHAR2(1),  
  CUST_DOB DATE,  
  ACCT_OUTSTANDING NUMBER(10,2),  
  ACCT_DATE_OPEN DATE,  
  ACCT_DATE_CLOSE DATE,  
  ACCT_COMPANY_NAME VARCHAR2(100),  
  ACCT_COMPANY_NAME_SHORT VARCHAR2(20),  
  ACCT_PARTNERSHIP_NAME VARCHAR2(100),  
  ACCT_PARTNERSHIP_NAME_SHORT VARCHAR2(20),  
  ACCT_NUMBER_OF_PARTNERS NUMBER(1),  
  ACCT_TRUST_NAME VARCHAR2(100),  
  ACCT_TRUST_NAME_SHORT VARCHAR2(20),  
  ACCT_TRUST_NUMBER NUMBER(10),  
  ACCT_ACN_NUMBER VARCHAR2(30),  
  CONSTRAINT CUST_PK PRIMARY KEY (CUST_ID)  
    USING INDEX PCTFREE 10 STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)  
    TABLESPACE USER_DATA,  
  CONSTRAINT CUST_GRP_PK PRIMARY KEY (CUST_GRP_ID)  
    USING INDEX PCTFREE 10 STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)  
    TABLESPACE USER_DATA)  
USING INDEX PCTFREE 10 STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)  
TABLESPACE USER_DATA)  
PARALLEL(DEGREE 1 INSTANCES 1) NOCACHE;

ALTER TABLE ST_CUSTOMER ADD CONSTRAINT  
SYS_C009165 CHECK (cust_sex in ('M', 'F', 'U', 'N/A'));

ALTER TABLE ST_CUSTOMER ADD CONSTRAINT  
SYS_C009166 CHECK (acct_registered IN ('Y', 'N'));

DROP TABLE ST_CUSTOMER_GRP CASCADE CONSTRAINTS;

CREATE TABLE ST_CUSTOMER_GRP (  
  CUST_GRP_ID NUMBER(4) NOT NULL,  
  CUST_GRP_NAME VARCHAR2(20),  
  CONSTRAINT CUST_GRP_PK PRIMARY KEY (CUST_GRP_ID)  
    USING INDEX PCTFREE 10 STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)  
    TABLESPACE USER_DATA)  
USING INDEX PCTFREE 10 STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)  
TABLESPACE USER_DATA)  
PARALLEL(DEGREE 1 INSTANCES 1) NOCACHE;
```

64
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

DROP TABLE ST_FORM CASCADE CONSTRAINTS;

CREATE TABLE ST_FORM (  
  FORM_ID NUMBER(4) NOT NULL,
  FORM_NAME VARCHAR2(100) NOT NULL,
  FORM_TITLE VARCHAR2(100) NOT NULL,
  FORM_WIDTH NUMBER(4) NOT NULL,
  FORM_HEIGHT NUMBER(4) NOT NULL,
  FORM_START DATE DEFAULT SYSDATE,
  FORM_END DATE,
  CONSTRAINT WIN_ID_PK PRIMARY KEY (FORM_ID)
  USING INDEX PCTFREE 10
  STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
  TABLESPACE USER_DATA)

TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

DROP TABLE ST_FORM_ITEM CASCADE CONSTRAINTS;

CREATE TABLE ST_FORM_ITEM (  
  FORM_ITEM_ID NUMBER(4) NOT NULL,
  FORM_ID NUMBER(4),
  ITEM_ID NUMBER(4),
  X_POS NUMBER(4),
  Y_POS NUMBER(4),
  LABEL VARCHAR2(50),
  FM_CALL VARCHAR2(50),
  WIDTH NUMBER(3),
  HEIGHT NUMBER(3),
  PROMPT VARCHAR2(50),
  ENABLED VARCHAR2(1),
  VISIBLE VARCHAR2(1),
  LOV VARCHAR2(50),
  START_DATE DATE,
  END_DATE DATE,
  CONSTRAINT WINIT_ID_PK PRIMARY KEY (FORM_ITEM_ID)
  USING INDEX PCTFREE 10
  STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
  TABLESPACE USER_DATA)

TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 16384 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

ALTER TABLE ST_FORM_ITEM ADD CONSTRAINT
SYS_C009220 CHECK (enabled IN ('Y', 'N'));

ALTER TABLE ST_FORM_ITEM ADD CONSTRAINT
SYS_C009221 CHECK (visible IN ('Y', 'N'));

DROP TABLE ST_ITEM CASCADE CONSTRAINTS;

CREATE TABLE ST_ITEM (  
  ITEM_ID NUMBER(4) NOT NULL,
ITEM_NAME VARCHAR2(100),
ITY_ID NUMBER(4) NOT NULL,
ITEM_ICON VARCHAR2(100),
START_DATE DATE,
END_DATE DATE,
CONSTRAINT ST_ITEM_ID_PK PRIMARY KEY (ITEM_ID)
USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;
DROP TABLE ST_ITEM_TYPE CASCADE CONSTRAINTS;
CREATE TABLE ST_ITEM_TYPE ( 
ITY_ID NUMBER(4) NOT NULL,
ITY_TYPE VARCHAR2(4),
ITY_DESCRIPTION VARCHAR2(50),
CONSTRAINT ST_ITTY_ID_PK PRIMARY KEY (ITY_ID)
USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;
DROP TABLE ST_ORDER CASCADE CONSTRAINTS;
CREATE TABLE ST_ORDER ( 
ORD_ID NUMBER(4) NOT NULL,
ORD_DATE DATE,
CUST_ID NUMBER(6) NOT NULL,
SALESMAN_ID NUMBER(6) NOT NULL,
SHIP_DATE DATE,
CONSTRAINT ORD l_PK PRIMARY KEY (ORD_ID)
USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;
DROP TABLE ST_ORDER_LINE CASCADE CONSTRAINTS;
CREATE TABLE ST_ORDER_LINE ( 
ORD_LINE_ID NUMBER(4) NOT NULL,
ORD_ID NUMBER(4) NOT NULL,
PROD_ID NUMBER(4) NOT NULL,
ACTUAL_PRICE NUMBER(8,2),
QUANTITY NUMBER(8),
ITEM_TOTAL NUMBER(8,2),
COMMISSION NUMBER(8,2),
CONSTRAINT ORD_LINE_PK PRIMARY KEY (ORD_LINE_ID, ORD_ID)
USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;
DROP TABLE ST_PRODUCT CASCADE CONSTRAINTS;

CREATE TABLE ST_PRODUCT (
    PROD_ID NUMBER(4) NOT NULL,
    PROD_NAME VARCHAR2(40),
    PROD_PRICE NUMBER(10),
    PROD_GRP_ID NUMBER(4),
    CONSTRAINT PROD_PK PRIMARY KEY (PROD_ID)
    USING INDEX PCTFREE 10
    STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50 )
    TABLESPACE USER_DATA)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50 )
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

DROP TABLE ST_PRODUCT_GRP CASCADE CONSTRAINTS;

CREATE TABLE ST_PRODUCT_GRP (
    PROD_GRP_ID NUMBER(4) NOT NULL,
    PROD_GRP_NAME VARCHAR2(20),
    CONSTRAINT PROD_GRP_PK PRIMARY KEY (PROD_GRP_ID)
    USING INDEX PCTFREE 10
    STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50 )
    TABLESPACE USER_DATA)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50 )
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

DROP TABLE ST_REF_ITEM CASCADE CONSTRAINTS;

CREATE TABLE ST_REF_ITEM (
    REF_ID NUMBER(4) NOT NULL,
    FROM_ITEM_ID NUMBER(4),
    TO_ITEM_ID NUMBER(4),
    REF_ITEM_VAL NUMBER(4),
    CONSTRAINT REF_ID_PK PRIMARY KEY (REF_ID)
    USING INDEX PCTFREE 10
    STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50 )
    TABLESPACE USER_DATA)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50 )
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

DROP TABLE ST_RULE CASCADE CONSTRAINTS;

CREATE TABLE ST_RULE (        
    RULE_ORDER NUMBER(4),
    PROD_GRP_ID NUMBER(4) NOT NULL,
    PROD_ID NUMBER(4) NOT NULL,
    CUST_GRP_ID NUMBER(4) NOT NULL,
    CUST_ID NUMBER(4) NOT NULL,
    SALESMAN_GRP_ID NUMBER(4) NOT NULL,
    SALESMAN_ID NUMBER(4) NOT NULL,
    COND_ID VARCHAR2(20) NOT NULL,
    CONDITION NUMBER(8,2),
    DESCRIPTION VARCHAR2(100),
    CONSTRAINT RULE_PK PRIMARY KEY (COND_ID, CUST_GRP_ID, CUST_ID,
SALESMAN_GRP_ID, SALESMAN_ID, PROD_GRP_ID, PROD_ID)
    USING INDEX PCTFREE 10
CREATE TABLE ST_RULE_PRECEDENCE (
  COND_ID VARCHAR2(20) NOT NULL,
  RULE_INDEX NUMBER(10) NOT NULL,
  PROD_GRP_ID NUMBER(1),
  PROD_ID NUMBER(1),
  CUST_GRP_ID NUMBER(1),
  CUST_ID NUMBER(1),
  SALESMAN_GRP_ID NUMBER(1),
  SALESMAN_ID NUMBER(1),
  DESCRIPTION VARCHAR2(100),
  CONSTRAINT ST_RULE_PRE_PK PRIMARY KEY (COND_ID, RULE_INDEX)
) USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

ALTER TABLE ST_RULE_PRECEDENCE ADD CONSTRAINT
SYS_C009198 CHECK (prod_grp_id BETWEEN 0 AND 1);

ALTER TABLE ST_RULE_PRECEDENCE ADD CONSTRAINT
SYS_C009199 CHECK (prod_id BETWEEN 0 AND 1);

ALTER TABLE ST_RULE_PRECEDENCE ADD CONSTRAINT
SYS_C009200 CHECK (cust_grp_id BETWEEN 0 AND 1);

ALTER TABLE ST_RULE_PRECEDENCE ADD CONSTRAINT
SYS_C009201 CHECK (cust_id BETWEEN 0 AND 1);

ALTER TABLE ST_RULE_PRECEDENCE ADD CONSTRAINT
SYS_C009202 CHECK (salesman_grp_id BETWEEN 0 AND 1);

ALTER TABLE ST_RULE_PRECEDENCE ADD CONSTRAINT
SYS_C009203 CHECK (salesman_id BETWEEN 0 AND 1);

CREATE UNIQUE INDEX ST_RULE_UK ON
ST_RULE_PRECEDENCE(PROD_GRP_ID, PROD_ID, CUST_GRP_ID, CUST_ID,
SALESMAN_GRP_ID, SALESMAN_ID)
TABLESPACE USER_DATA PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50);

DROP TABLE ST_SALESMAN CASCADE CONSTRAINTS;

CREATE TABLE ST_SALESMAN (
  SALESMAN_ID NUMBER(4) NOT NULL,
  SALESMAN_GIVEN_NAMES VARCHAR2(40),
  SALESMAN_SURNAME VARCHAR2(40),
);
SELECT
SALESMAN_TITLE VARCHAR2(6),
SALESMAN_ADDRESS_1 VARCHAR2(40),
SALESMAN_ADDRESS_2 VARCHAR2(40),
SALESMAN_ADDRESS_3 VARCHAR2(40),
SALESMAN_SUBURB VARCHAR2(20),
SALESMAN_POSTCODE VARCHAR2(10),
SALESMAN_STATE VARCHAR2(4),
SALESMAN_COUNTRY VARCHAR2(20),
SALESMAN_PHONE NUMBER(14),
SALESMAN_FAX NUMBER(14),
SALESMAN_CR_RATING VARCHAR2(10),
SALESMAN_GRP_ID NUMBER(4),
CONSTRAINT SALESMAN_PK PRIMARY KEY (SALESMAN_ID)
USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA);

DROP TABLE ST_SALESMAN_GRP CASCADE CONSTRAINTS;
CREATE TABLE ST_SALESMAN_GRP
(SALESMAN_GRP_ID NUMBER(4) NOT NULL,
SALESMAN_GRP_NAME VARCHAR2(20),
CONSTRAINT SALESMAN_GRP_PK PRIMARY KEY (SALESMAN_GRP_ID)
USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA);

DROP TABLE ST_SEQ CASCADE CONSTRAINTS;
CREATE TABLE ST_SEQ
(SEQ_NAME VARCHAR2(40) NOT NULL,
SEQ_CURR_VALUE NUMBER(8) NOT NULL,
SEQ_MIN_VALUE NUMBER(8) NOT NULL,
SEQ_MAX_VALUE NUMBER(8) NOT NULL)
TABLESPACE USER_DATA PCTUSED 40 PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
PARALLEL (DEGREE 1 INSTANCES 1) NOCACHE;

DROP TABLE ST_SYSTEM_PARAMETERS CASCADE CONSTRAINTS;
CREATE TABLE ST_SYSTEM_PARAMETERS
(PAR_ID NUMBER(4) NOT NULL,
PAR_NAME VARCHAR2(100),
PAR_VALUE NUMBER(4),
START_DATE DATE,
END_DATE DATE,
CONSTRAINT ST__FAR_PK PRIMARY KEY (PAR_ID)
USING INDEX PCTFREE 10
STORAGE(INITIAL 10240 NEXT 10240 PCTINCREASE 50)
TABLESPACE USER_DATA);

69
DROP TABLE TEM CASCADE CONSTRAINTS;

ALTER TABLE ST_CUSTOMER ADD CONSTRAINT CUST_FK
FOREIGN KEY (CUST_GRP_ID)
REFERENCES ST_CUSTOMER_GRP (CUST_GRP_ID);

ALTER TABLE ST_FORM_ITEM ADD CONSTRAINT WINIT_WIN_FK
FOREIGN KEY (FORM_ID)
REFERENCES ST_FORM (FORM_ID);

ALTER TABLE ST_FORM_ITEM ADD CONSTRAINT WINIT_ITM_FK
FOREIGN KEY (ITEM_ID)
REFERENCES ST_ITEM (ITEM_ID);

ALTER TABLE ST_ITEM ADD CONSTRAINT ST_ID_FK
FOREIGN KEY (ITTY_ID)
REFERENCES ST_ITEM_TYPE (ITTY_ID);

ALTER TABLE ST_ORDER ADD CONSTRAINT ORD_CUST_FK
FOREIGN KEY (CUST_ID)
REFERENCES ST_CUSTOMER (CUST_ID);

ALTER TABLE ST_ORDER ADD CONSTRAINT ORD_SALESMAN_FK
FOREIGN KEY (SALESMAN_ID)
REFERENCES ST_SALESMAN (SALESMAN_ID);

ALTER TABLE ST_ORDER_LINE ADD CONSTRAINT ORD_LINE_ORD_FK
FOREIGN KEY (ORD_ID)
REFERENCES ST_ORDER (ORD_ID);

ALTER TABLE ST_ORDER_LINE ADD CONSTRAINT ORD_LINE_PROD_FK
FOREIGN KEY (PROD_ID)
REFERENCES ST_PRODUCT (PROD_ID);

ALTER TABLE ST_PRODUCT ADD CONSTRAINT PROD_FK
FOREIGN KEY (PROD_GRP_ID)
REFERENCES ST_PRODUCT_GRP (PROD_GRP_ID);

ALTER TABLE ST_REF_ITEM ADD CONSTRAINT REF_TO_ITEM_FK
FOREIGN KEY (TO_ITEM_ID)
REFERENCES ST_FORM_ITEM (FORM_ITEM_ID);

ALTER TABLE ST_REF_ITEM ADD CONSTRAINT REF_FROM_ITEM_FK
FOREIGN KEY (FROM_ITEM_ID)
REFERENCES ST_FORM_ITEM (FORM_ITEM_ID);

ALTER TABLE ST_RULE ADD CONSTRAINT CUST_GRP_ID_FK
FOREIGN KEY (CUST_GRP_ID)
REFERENCES ST_CUSTOMER_GRP (CUST_GRP_ID);

ALTER TABLE ST_RULE ADD CONSTRAINT CUST_ID_FK
FOREIGN KEY (CUST_ID)
REFERENCES ST_CUSTOMER (CUST_ID);

ALTER TABLE ST_RULE ADD CONSTRAINT SALESMAN_GRP_ID_FK
FOREIGN KEY (SALESMAN_GRP_ID)
REFERENCES ST_SALESMAN_GRP (SALESMAN_GRP_ID);

ALTER TABLE ST_RULE ADD CONSTRAINT SALESMAN_ID_FK
FOREIGN KEY (SALESMAN_ID)
REFERENCES ST_SALESMAN (SALESMAN_ID);

ALTER TABLE ST_RULE ADD CONSTRAINT SALESMAN_ID_FK
FOREIGN KEY (SALESMAN_ID)
REFERENCES ST_SALESMAN (SALESMAN_ID);

ALTER TABLE ST_RULE ADD CONSTRAINT PROD_GRP_ID_FK
FOREIGN KEY (PROD_GRP_ID)
REFERENCES ST_PRODUCT_GRP (PROD_GRP_ID);

ALTER TABLE ST_RULE ADD CONSTRAINT PROD_ID_FK
FOREIGN KEY (PROD_ID)
REFERENCES ST_PRODUCT (PROD_ID);

ALTER TABLE ST_SALESMAN ADD CONSTRAINT SALESMAN_FK
FOREIGN KEY (SALESMAN_GRP_ID)
REFERENCES ST_SALESMAN_GRP (SALESMAN_GRP_ID);
Appendix B – Oracle PL/SQL Packages

PACKAGE Item_Man IS
    PROCEDURE Show_Buttons ;
    PROCEDURE Set_Up_Items (p_val IN NUMBER , p_item IN VARCHAR2) ;
    PROCEDURE Size_Window ;
    PROCEDURE Call_New_Form (p_form IN VARCHAR2) ;
    PROCEDURE Update_Properties (p_blk_name IN VARCHAR2) ;
    PROCEDURE New_Block (p_query_blk IN VARCHAR2 , p_query IN VARCHAR2) ;
    PROCEDURE Hide_All_Items (p_blk_name IN VARCHAR2) ;
    FUNCTION Calc_Commission (p_prod_grp_id IN NUMBER , p_prod_id IN NUMBER , p_cust_grp_id IN NUMBER , p_cust_id IN NUMBER , p_sales_grp_id IN NUMBER , p_sales_id IN NUMBER , p_cond_id IN NUMBER) RETURN NUMBER ;
END;

PACKAGE BODY Item_Man IS
CURSOR itms (item_type IN VARCHAR2) IS
SELECT i.item_name , i.item_icon FROM st_form_item fi , st_form f , st_item i , st_item_type it WHERE f.form_name = :GLOBAL.gv_form AND f.form_id = fi.form_id AND fi.item_id = i.item_id AND i.itty_id = it.itty_id AND it.itty_type = item_type AND i.end_date IS NULL ORDER BY i.item_id ;
gv_alert NUMBER ;

PROCEDURE Show_buttons IS

**************************************************************************
***
* Procedure to select the buttons that should appear on the button bar when *
* a form is first opened. Dynamically places the buttons on the form       *
**************************************************************************
***/
-- Procedure Show_Buttons is used to --
-- Set up the navigation buttons --
  lv_button_count NUMBER := 1 ;
  lv_x_pos NUMBER := st_pkg.get_parameter_value ('BUTTON_START_X') ;
  lv_y_pos NUMBER := st_pkg.get_parameter_value ('BUTTON_START_Y') ;
  lv_right_margin NUMBER := st_pkg.get_parameter_value ('BUTTON_RIGHT_MARGIN') ;

72
lv_bottom_margin NUMBER := st_pkg.get_parameter_value (BUTTON_BOTTOM_MARGIN) ;
lv_mod NUMBER := st_pkg.get_parameter_value (MAX_ROW_LENGTH) ;
lv_form_height NUMBER := 0 ;
lv_form_width NUMBER := 0 ;
lv_inc_width BOOLEAN := TRUE ;
lv_last_height NUMBER := 0 ;
lv_last_width NUMBER := 0 ;
lv_count NUMBER := 0 ;
BEGIN
FOR nav_itms IN itms (B) LOOP
  -- For each item in the cursor, display it and set the icon --
  GO_ITEM (nav_itms.item_name) ;
  IF lv_button_count = 1 THEN
    lv_form_width := lv_form_width + GET_ITEM_PROPERTY (nav_itms.item_name, WIDTH) ;
  END IF ;

  SET_ITEM_PROPERTY (nav_itms.item_name, X_POS, lv_x_pos) ;
  SET_ITEM_PROPERTY (nav_itms.item_name, Y_POS, lv_y_pos) ;
  SET_ITEM_PROPERTY (nav_itms.item_name, VISIBLE, PROPERTY_TRUE) ;
  SET_ITEM_PROPERTY (nav_itms.item_name, ENABLED, PROPERTY_TRUE) ;
  SET_ITEM_PROPERTY (nav_itms.item_name, ICON_NAME, nav_itms.item_icon) ;

  -- Update the x and y coordinates for the next item

  IF lv_inc_width = TRUE THEN
    lv_count := lv_count + 1 ;
  END IF ;
  IF MOD(lv_button_count, lv_mod) = 0 THEN -- Time to move to the next row
    lv_x_pos := st_pkg.get_parameter_value (BUTTON_START_X) ;
    lv_y_pos := lv_y_pos + GET_ITEM_PROPERTY (nav_itms.item_name, HEIGHT) ;
    lv_form_height := lv_form_height + GET_ITEM_PROPERTY (nav_itms.item_name, HEIGHT) ;
    lv_inc_width := FALSE ;
  ELSE
    lv_x_pos := lv_x_pos + GET_ITEM_PROPERTY (nav_itms.item_name, WIDTH) ;
  END IF ;
  IF lv_inc_width = TRUE THEN
    lv_form_width := lv_form_width + GET_ITEM_PROPERTY (nav_itms.item_name, WIDTH) ;
  END IF ;
  lv_button_count := lv_button_count + 1 ; -- Increment the counter --
  lv_last_height := GET_ITEM_PROPERTY (nav_itms.item_name, HEIGHT) ;
  lv_last_width := GET_ITEM_PROPERTY (nav_itms.item_name, WIDTH) ;
END LOOP ;
IF MOD(lv_button_count - 1, lv_mod) /= 0 THEN
  lv_form_height := lv_form_height + lv_last_height ;
END IF ;
-- Set up the height and width of the form based on the number of
-- displayed buttons.
IF st_pkg.get_button_count (:GLOBAL.gv_form) = 1 THEN
  lv_form_width := lv_form_width / 2 ;
ELSIF lv_count < lv_mod THEN
  lv_form_width := lv_form_width - lv_last_width ;
  lv_form_height := lv_form_height + lv_last_height ;
END IF;
SET_WINDOWPROPERTY ('WIN_BUTTON', WINDOW_SIZE, lv_form_width +
lv_right_margin, lv_form_height + lv_bottom_margin);
SET_CANVASPROPERTY ('CNV_BUTTON', CANVAS_SIZE, lv_form_width +
lv_right_margin, lv_form_height + lv_bottom_margin);

END Show_Buttons;

PROCEDURE Set_Up_Items (p_val IN NUMBER
, p_item IN VARCHAR2) IS

lv_item_ht VARCHAR2(100);
CURSOR cur_itm IS

SELECT i.item_name
 , i.iitty_type
 , fi.fm_call
FROM st_item i
 , st_form_item fi
 , st_ref_item ri
 , st_item_type it
WHERE i.item_id = fi.item_id
AND fi.form_item_id = ri.to_item_id
AND i.iitty_id = it.iitty_id
AND ri.ref_id IN (SELECT ri2.ref_id
FROM st_ref_item ri2
 , st_form_item fi2
 , st_item i2
WHERE ri2.ref_item_val = p_val
AND ri2.from_item_id = fi2.form_item_id
AND fi2.item_id = i2.item_id
AND i2.item_name = p_item);

BEGIN

   Hide_All_Items (:SYSTEM.trigger_block);
   FOR c_itm IN cur_itm LOOP
      IF c_itm.iitty_type IN ( 'T' , 'L' ) THEN
         SET_ITEMPROPERTY (c_itm.item_name, VISIBLE, PROPERTY_TRUE);
         SET_ITEMPROPERTY (c_itm.item_name, ENABLED, PROPERTY_TRUE);
      ELSE
         SET_RADIO_BUTTONPROPERTY (c_itm.item_name, c_itm.fm_call, VISIBLE,
PROPERTY_TRUE);
         SET_RADIO_BUTTONPROPERTY (c_itm.item_name, c_itm.fm_call, ENABLED,
PROPERTY_TRUE);
      END IF;
   END LOOP;
END Set_Up_Items;

PROCEDURE Hide_All_Items (p_blk_name IN VARCHAR2) IS

lv_prev_item VARCHAR2(100);

BEGIN
   Hide_All_Items (:SYSTEM.trigger_block);
   FOR c_itm IN cur_itm LOOP
      IF c_itm.iitty_type IN ( 'T' , 'L' ) THEN
         SET_ITEMPROPERTY (c_itm.item_name, VISIBLE, PROPERTY_TRUE);
         SET_ITEMPROPERTY (c_itm.item_name, ENABLED, PROPERTY_TRUE);
      ELSE
         SET_RADIO_BUTTONPROPERTY (c_itm.item_name, c_itm.fm_call, VISIBLE,
PROPERTY_TRUE);
         SET_RADIO_BUTTONPROPERTY (c_itm.item_name, c_itm.fm_call, ENABLED,
PROPERTY_TRUE);
      END IF;
   END LOOP;
END Hide_All_Items;
lv_last_item VARCHAR2(100) := GET_BLOCK_PROPERTY(p_blk_name, LAST_ITEM);
lv_next_item VARCHAR2(100);
BEGIN
  GO_ITEM (GET_BLOCK_PROPERTY(p_blk_name, FIRST_ITEM));
  LOOP
    lv_next_item := GET_ITEM_PROPERTY (:SYSTEM.current_item, NEXTITEM);
    SET_ITEM_PROPERTY (lv_next_item, VISIBLE, PROPERTY_TRUE);
    SET_ITEM_PROPERTY (lv_next_item, ENABLED, PROPERTY_TRUE);
    GO_ITEM (lv_next_item);
    EXIT WHEN :SYSTEM.current_item = lv_last_item;
  END LOOP;
  GO_ITEM (GET_BLOCK_PROPERTY(p_blk_name, FIRST_ITEM));
  GO_ITEM (GET_ITEM_PROPERTY (:SYSTEM.current_item, NEXTITEM));
  lv_prev_item := :SYSTEM.current_item;
  LOOP
    GO_ITEM (GET_ITEM_PROPERTY (:SYSTEM.current_item, NEXTITEM)) ;
    SET_ITEM_PROPERTY (lv_prev_item, VISIBLE, PROPERTY_FALSE);
    SET_ITEM_PROPERTY (lv_prev_item, ENABLED, PROPERTY_FALSE);
    lv_prev_item := :SYSTEM.current_item;
    EXIT WHEN :SYSTEM.current_item = lv_last_item;
  END LOOP;
  GO_ITEM (GET_BLOCK_PROPERTY(p_blk_name, FIRST_ITEM));
  SET_ITEM_PROPERTY (lv_last_item, VISIBLE, PROPERTY_FALSE);
  SET_ITEM_PROPERTY (lv_last_item, ENABLED, PROPERTY_FALSE);
END Hide_All_Items;
/******************** ****** ***************************** ************* !
PROCEDURE Size_Window IS
  lv_width NUMBER(4) := st_pkg.get_window_width(:GLOBAL.gv_form);
  lv_height NUMBER(4):= st_pkg.get_window_height(:GLOBAL.gv_form);
  lv_title VARNCHAR2(100) := st_pkg.get_window_title (:GLOBAL.gv_form);
BEGIN
  Set_Window_Property ('WIN_MAIN', TITLE, lv_title);
  Set_Window_Property ('WIN_MAIN', WINDOW_SIZE, lv_width , lv_height);
  Set_Canvas_Property ('CNV_MAIN', CANVAS_SIZE, lv_width , lv_height);
  :flex_dummy.form_title := lv_title;
END size_window;
/******************** ****** ***************************** ************* !
PROCEDURE Update_Properties (p_blk_name IN VARCHAR2) IS
  lv_invalid_item EXCEPTION ;
  CURSOR itms IS
    SELECT i.item_name
    , fi.x_pos
    , fi.y_pos
    , fi.label
    , fi.width
    , fi.height
    FROM

FROM  st_item i
    , st_form f
    , st_form_item fi
    , st_item_type it
WHERE  fi.form_id = f.form_id
AND    f.form_name = :GLOBAL.gv_form
AND    fi.item_id = i.item_id
AND    i.itty_id = it.itty_id
AND    SUBSTR(i.item_name, 1, INSTR(i.item_name, '.') - 1) = p_blk_name ;
BEGIN
FOR cur_itms IN itms LOOP
    -- Check to see if the item exists on the Form. If not raise an exception
    it_id := Find_Item(cur_itms.item_name);
    IF Id_Null(it_id) THEN
        RAISE lv_invalid_item ;
    END IF ;
    --
    IF cur_itms.x_pos IS NOT NULL THEN
        IF cur_itms.itty_type IN (T', L') THEN
            SET_ITEM_PROPERTY (cur_itms.item_name, X_POS, cur_itms.x_pos) ;
        ELSE
            SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, X_POS, cur_itms.x_pos) ;
        END IF ;
    END IF ;
    --
    IF cur_itms.y_pos IS NOT NULL THEN
        IF cur_itms.itty_type IN (T', L') THEN
            SET_ITEM_PROPERTY (cur_itms.item_name, Y_POS, cur_itms.y_pos) ;
        ELSE
            SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, Y_POS, cur_itms.y_pos) ;
        END IF ;
    END IF ;
    --
    IF cur_itms.width IS NOT NULL THEN
        IF cur_itms.itty_type IN (T', L') THEN
            SET_ITEM_PROPERTY (cur_itms.item_name, WIDTH, cur_itms.width) ;
        ELSE
            SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, WIDTH, cur_itms.width) ;
        END IF ;
    END IF ;
    --
    IF cur_itms.height IS NOT NULL THEN
        IF cur_itms.itty_type IN (T', L') THEN
            SET_ITEM_PROPERTY (cur_itms.item_name, HEIGHT, cur_itms.height) ;
        ELSE
            SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, HEIGHT, cur_itms.height) ;
        END IF ;
    END IF ;
END LOOP ;
SET_ITEM_PROPERTY (cur_itms.item_name, HEIGHT, cur_itms.height);
ELSE
  SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, HEIGHT, cur_itms.height);
END IF;
END IF;

--
IF cur_itms.visible = 'Y' THEN
  IF cur_itms.itty_type IN ( 'T', 'L' ) THEN
    SET_ITEM_PROPERTY (cur_itms.item_name, VISIBLE, PROPERTY_TRUE);
  ELSE
    SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, VISIBLE, PROPERTY_TRUE);
  END IF;
ELSE
  IF cur_itms.itty_type IN ( 'T', 'L' ) THEN
    SET_ITEMPROPERTY (cur_itms.item_name, VISIBLE, PROPERTY_FALSE);
  ELSE
    SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, VISIBLE, PROPERTY_FALSE);
  END IF;
END IF;

--
IF cur_itms.enabled = 'Y' THEN
  IF cur_itms.itty_type IN ( 'T', 'L' ) THEN
    SET_ITEM_PROPERTY (cur_itms.item_name, ENABLED, PROPERTY_TRUE);
  ELSE
    SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, ENABLED, PROPERTY_TRUE);
  END IF;
ELSE
  IF cur_itms.itty_type IN ( 'T', 'L' ) THEN
    SET_ITEM_PROPERTY (cur_itms.item_name, ENABLED, PROPERTY_FALSE);
  ELSE
    SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, ENABLED, PROPERTY_FALSE);
  END IF;
END IF;

--
IF cur_itms.label IS NOT NULL THEN
  IF cur_itms.itty_type IN ( 'T', 'L' ) THEN
    SET_ITEM_PROPERTY (cur_itms.item_name, PROMPT_TEXT, cur_itms.label);
  ELSE
    SET_RADIO_BUTTON_PROPERTY (cur_itms.item_name, cur_itms.fm_call, LABEL, cur_itms.fm_call);
  END IF;
END IF;
END IF;

--
EXCEPTION
WHEN lv_invalid_item THEN
  Set_Alert_Property('al_error', ALERT_MESSAGE_TEXT, cur_itms.item_name);
  gv_alert := SHOW_ALERT('al_error');
WHEN OTHERS THEN
  RAISE FORM_TRIGGER_FAILURE;
END;

77
PROCEDURE Set_Up_Dummy (p_blk_name IN VARCHAR2) IS
PROCEDURE New_Block (p_query_blk IN VARCHAR2, p_query IN VARCHAR2) IS
FUNCTION Calc_Commission (p_prod_grp_id IN NUMBER, p_prod_id IN NUMBER, p_cust_grp_id IN NUMBER, p_cust_id IN NUMBER, p_sales_grp_id IN NUMBER, p_sales_id IN NUMBER, p_cond_id IN NUMBER) RETURN NUMBER IS
, p_cust_grp_id
, p_cust_id
, p_sales_grp_id
, p_sales_id
, p_cond_id);
END Calc_Commission;
END;
CREATE OR REPLACE PACKAGE st_pkg IS
FUNCTION get_window_height (p_window_name VARCHAR2) RETURN NUMBER;
FUNCTION get_window_width (p_window_name VARCHAR2) RETURN NUMBER;
FUNCTION get_window_title (p_title VARCHAR2) RETURN VARCHAR2;
PROCEDURE get_button_coords (p_min_width IN OUT NUMBER
, p_max_width IN OUT NUMBER
, p_min_height IN OUT NUMBER
, p_max_height IN OUT NUMBER
, p_form_name IN VARCHAR2);
FUNCTION get_parameter_value (p_param IN VARCHAR2) RETURN NUMBER;
FUNCTION get_button_count (p_form_name IN VARCHAR2) RETURN NUMBER;
PROCEDURE Get_Item_Property (p_cur_item IN VARCHAR2
, p_form_name IN VARCHAR2
, p_x_pos IN OUT NUMBER
, p_y_pos IN OUT NUMBER
, p_label IN OUT VARCHAR2
, p_width IN OUT NUMBER
, p_height IN OUT NUMBER
, p_enabled IN OUT VARCHAR2
, p_visible IN OUT VARCHAR2
, p_lov IN OUT VARCHAR2
, p_id IN OUT NUMBER);
FUNCTION get_item_type (p_item_id IN NUMBER) RETURN VARCHAR2;
FUNCTION Get_Customer_Grp (p_grp_id IN NUMBER) RETURN VARCHAR2;
FUNCTION Get_Cust_Name (p_id IN NUMBER) RETURN VARCHAR2;
FUNCTION Get_Sales_Name (p_id IN NUMBER) RETURN VARCHAR2;
FUNCTION Get_Product_Name (p_id IN NUMBER) RETURN VARCHAR2;
END st_pkg;
/
CREATE OR REPLACE PACKAGE st_pkg IS
FUNCTION get_window_height (p_window_name IN st_form.form_name%TYPE) RETURN NUMBER;
FUNCTION get_window_width (p_window_name IN st_form.form_name%TYPE) RETURN NUMBER;
FUNCTION get_window_title (p_title IN st_form.form_name%TYPE) RETURN VARCHAR2;
PROCEDURE get_button_coords (p_min_width IN OUT st_form_item.x_pos%TYPE
, p_max_width IN OUT st_form_item.x_pos%TYPE
, p_min_height IN OUT st_form_item.y_pos%TYPE
, p_max_height IN OUT st_form_item.y_pos%TYPE
, p_form_name IN st_form.form_name%TYPE);
FUNCTION get_parameter_value (p_param IN st_system_parameters.par_name%TYPE) RETURN NUMBER;
FUNCTION get_button_count (p_form_name IN st_form.form_name%TYPE) RETURN NUMBER;
PROCEDURE Get_Item_Property (p_cur_item IN VARCHAR2
, p_form_name IN VARCHAR2
, p_x_pos IN OUT NUMBER
, p_y_pos IN OUT NUMBER
, p_label IN OUT VARCHAR2
, p_width IN OUT NUMBER
, p_height IN OUT NUMBER
, p_enabled IN OUT VARCHAR2
, p_visible IN OUT VARCHAR2
, p_lov IN OUT VARCHAR2
, p_id IN OUT NUMBER);
80
CREATE OR REPLACE PACKAGE BODY st_pkg IS

FUNCTION get_window_width (p_window_name st_form.form_name%TYPE) RETURN NUMBER IS
CURSOR cur_width IS
SELECT form_width
FROM st_form
WHERE form_name = p_window_name;

LV_WIDTH st_form.form_width%TYPE;
BEGIN
OPEN cur_width;
FETCH cur_width INTO lv_width;
CLOSE cur_width;
RETURN (lv_width);
END get_window_width;

FUNCTION get_window_height (p_window_name st_form.form_name%TYPE) RETURN NUMBER IS
LV_HEIGHT st_form.form_height%TYPE;
BEGIN
SELECT form_height
INTO lv_height
FROM st_form
WHERE form_name = p_window_name;

RETURN (lv_height);
END get_window_height;

FUNCTION get_window_title (p_title IN st_form.form_name%TYPE) RETURN VARCHAR2 IS
BEGIN
RETURN p_title;
END get_window_title;

END st_pkg;
/
lv_title st_form.form_title%TYPE;
BEGIN
    SELECT form_title
    INTO lv_title
    FROM st_form
    WHERE form_name = p_title;
--
    RETURN (lv_title);
END get_window_title;

PROCEDURE get_button_coords (p_min_width IN OUT st_form_item.x_pos%TYPE
, p_max_width IN OUT st_form_item.x_pos%TYPE
, p_min_height IN OUT st_form_item.y_pos%TYPE
, p_max_height IN OUT st_form_item.y_pos%TYPE
, p_form_name IN st_form.form_name%TYPE) IS
CURSOR win_cord IS
    SELECT min(fi.x_pos)
    , max(fi.x_pos)
    , min(fi.y_pos)
    , max(fi.y_pos)
    FROM st_form_item fi
    , st_form f
    , st_item i
    WHERE fi.item_id = i.item_id
    AND fi.form_id = f.form_id
    AND f.form_name = p_form_name
    AND i.item_name LIKE 'NAV_BUTTONS.BUTTON_%';
BEGIN
    OPEN win_cord;
    FETCH win_cord INTO p_min_width
    , p_max_width
    , p_min_height
    , p_max_height;
    CLOSE win_cord;
END get_button_coords;

FUNCTION get_parameter_value (p_param IN st_system_parameters.par_name%TYPE)
RETURN NUMBER IS
FUNCTION to return a user defined parameter
END get_parameter_value;

lv_param st_system_parameters.par_value%TYPE;
BEGIN
    SELECT par_value
    INTO lv_param
FROM  st_system_parameters
WHERE  par_name = p_param ;

--
RETURN (lv_param) ;
END get_parameter_value ;

FUNCTION get_button_count (p_form_name IN st_form.form_name%TYPE) RETURN NUMBER IS

FUNCTION to count the number of buttons that are displayed for a form

lv_count NUMBER ;
CURSOR itms (item_type IN st_form.form_name%TYPE) IS
SELECT COUNT( l)
FROM st_form_item fi
, st_form f
, st_item i
, st_item_type it
WHERE f.form_name = p_form_name
AND  f.form_id = fi.form_id
AND  fi.item_id = i.item_id
AND  i.itty_id = it.itty_id
AND  it.itty_type = item_type ;
BEGIN
OPEN itms (B) ;
FETCH itms INTO lv_count ;
CLOSE itms ;
RETURN (lv_count) ;
END get_button_count ;

PROCEDURE Get_Item_Property (p_cur_item IN VARCHAR2
, p_form_name IN VARCHAR2
, p_x_pos IN OUT NUMBER
, p_y_pos IN OUT NUMBER
, p_label IN OUT VARCHAR2
, p_width IN OUT NUMBER
, p_height IN OUT NUMBER
, p_enabled IN OUT VARCHAR2
, p_visible IN OUT VARCHAR2
, p_lov IN OUT VARCHAR2
, p_id IN OUT NUMBER) IS

PROCEDURE to retrieve all the item properties for items on a form

CURSOR itms IS
SELECT fi.x_pos
, fi.y_pos
, fi.label
, fi.width
, fi.height
, fi.enabled

83
FROM st_item i, st_form f, st_form_item fi
WHERE fi.form_id = f.form_id
AND f.form_name = p_form_name
AND fi.item_id = i.item_id
AND i.item_name = p_cur_item;
BEGIN
OPEN itms;
FETCH itms INTO p_x_pos, p_y_pos, p_label, p_width, p_height, p_enabled, p_visible, p_lov, p_id;
CLOSE itms;
END;

FUNCTION get_item_type (p_item_id IN st_item.item_id%TYPE) RETURN VARCHAR2
IS
lv_item_type st_item_type.itty_description%TYPE;
BEGIN
SELECT it.itty_description INTO lv_item_type
FROM st_item_type it, st_item i
WHERE i.item_id = p_item_id
AND i.itty_id = it.itty_id;
RETURN (lv_item_type);
END get_item_type;

FUNCTION Get_Customer_Grp (p_grp_id IN st_customer_grp.cust_grp_id%TYPE) RETURN VARCHAR2
IS
lv_cust_name st_customer_grp.cust_grp_name%TYPE;
BEGIN
SELECT cust_grp_name INTO lv_cust_name
FROM st_customer_grp
WHERE cust_grp_id = p_grp_id;
RETURN lv_cust_name;
END Get_Customer_Grp;
WHERE cust_grp_id = p_grp_id;
--
RETURN lv_cust_name;
END Get_Customer_Grp;

/*-----------------------------------------*/

FUNCTION Get_Cust_Name (p_id IN st_customer.cust_id%TYPE) RETURN VARCHAR2 IS

/* Function to get the customers name */
lv_name VARCHAR2(100);
BEGIN
SELECT cust_surname||' '||cust_given_names
INTO lv_name
FROM st_customer
WHERE cust_id = p_id;
--
RETURN lv_name;
END Get_Cust_Name;

/*-----------------------------------------*/

FUNCTION Get_Sales_Name (p_id IN st_salesman.salesman_id%TYPE) RETURN VARCHAR2 IS

/* Function to get the salesman name */
lv_name VARCHAR2(100);
BEGIN
SELECT salesman_surname||' '||salesman_given_names
INTO lv_name
FROM st_salesman
WHERE salesman_id = p_id;
--
RETURN lv_name;
END Get_Sales_Name;

/*-----------------------------------------*/

FUNCTION Get_Product_Name (p_id IN st_product.prod_id%TYPE) RETURN VARCHAR2 IS

/* Function to get the product group */
lv_name VARCHAR2(100);
BEGIN
SELECT prod_name
INTO lv_name
FROM st_product
WHERE prod_id = p_id;
--
RETURN lv_name;
END Get_Product_Name;
END st_pkg;
/

CREATE OR REPLACE PACKAGE st_comm IS

FUNCTION Calculate_Commission (p_prod_grp_id IN NUMBER
, p_prod_id IN NUMBER
, p_cust_grp_id IN NUMBER
, p_cust_id IN NUMBER
, p_sales_grp_id IN NUMBER
, p_sales_id IN NUMBER
, p_cond_id IN NUMBER) RETURN NUMBER;

FUNCTION Display_Valid_LOV (p_lov IN st_item.item_name%TYPE
, p_form IN st_form.form_name%TYPE) RETURN VARCHAR2;

FUNCTION Get_Form_Size (p_value IN st_system_parameters.par_name%TYPE) RETURN NUMBER;

FUNCTION Get_Cust_Grp (p_cust_id IN st_customer.cust_id%TYPE) RETURN NUMBER;

FUNCTION Get_Sales_Grp (p_sales_id IN st_salesman.salesman_id%TYPE) RETURN NUMBER;

FUNCTION Get_Prod_Grp (p_prod_id IN st_product.prod_id%TYPE) RETURN NUMBER;

FUNCTION Get_Sequence_Number (p_seq_name IN VARCHAR2) RETURN NUMBER;

END st_comm;
/
CREATE OR REPLACE PACKAGE BODY st_comm IS

!/***************************************************************************************
FUNCTION Calculate_Commission (p_prod_grp_id IN NUMBER
, p_prod_id IN NUMBER
, p_cust_grp_id IN NUMBER
, p_cust_id IN NUMBER
, p_sales_grp_id IN NUMBER
, p_sales_id IN NUMBER
, p_cond_id IN NUMBER) RETURN NUMBER IS

CURSOR chk_cond IS
SELECT count( l)
FROM st_rule_precedence
WHERE cond_id = p_cond_id ;

CURSOR cur_rule IS
SELECT *
FROM st_rule
ORDER by rule_order ;

CURSOR cur_rule_precedence IS
SELECT *
FROM st_rule_precedence
ORDER BY rule_index ;

lv_prod_grp_id NUMBER := 0 ;
lv_prod_id NUMBER := 0 ;
lv_cust_grp_id NUMBER := 0 ;
lv_cust_id NUMBER := 0 ;
lv_sales_grp_id NUMBER := 0 ;
lv_sales_id NUMBER := 0 ;

*****************************************************************************
* Function to calculate the amount of commission owing
*****************************************************************************

CURSOR chk_cond IS
SELECT count( l)
FROM st_rule_precedence
WHERE cond_id = p_cond_id ;

CURSOR cur_rule IS
SELECT *
FROM st_rule
ORDER by rule_order ;

CURSOR cur_rule_precedence IS
SELECT *
FROM st_rule_precedence
ORDER BY rule_index ;

lv_prod_grp_id NUMBER := 0 ;
lv_prod_id NUMBER := 0 ;
lv_cust_grp_id NUMBER := 0 ;
lv_cust_id NUMBER := 0 ;
lv_sales_grp_id NUMBER := 0 ;
lv_sales_id NUMBER := 0 ;
lv_exit  VARCHAR2(10) := 'FALSE';
lv_cond  NUMBER  := 0;

BEGIN
  FOR st_rule_precedence IN cur_rule_precedence LOOP
    --
    IF st_rule_precedence.prod_grp_id = 1 THEN
      lv_prod_grp_id := p_prod_grp_id;
    ELSE
      lv_prod_grp_id := 0;
    END IF;
    --
    IF st_rule_precedence.prod_id = 1 THEN
      lv_prod_id := p_prod_id;
    ELSE
      lv_prod_id := 0;
    END IF;
    --
    IF st_rule_precedence.cust_grp_id = 1 THEN
      lv_cust_grp_id := p_cust_grp_id;
    ELSE
      lv_cust_grp_id := 0;
    END IF;
    --
    IF st_rule_precedence.cust_id = 1 THEN
      lv_cust_id := p_cust_id;
    ELSE
      lv_cust_id := 0;
    END IF;
    --
    IF st_rule_precedence.salesman_grp_id = 1 THEN
      lv_sales_grp_id := p_sales_grp_id;
    ELSE
      lv_sales_grp_id := 0;
    END IF;
    --
    IF st_rule_precedence.salesman_id = 1 THEN
      lv_sales_id := p_sales_id;
    ELSE
      lv_sales_id := 0;
    END IF;
    FOR st_rule IN cur_rule LOOP
      IF (lv_prod_grp_id = st_rule.prod_grp_id) AND
         (lv_prod_id   = st_rule.prod_id) AND
         (lv_cust_grp_id = st_rule.cust_grp_id) AND
         (lv_cust_id   = st_rule.cust_id) AND
         (lv_sales_grp_id = st_rule.salesman_grp_id) AND
         (lv_sales_id   = st_rule.salesman_id) AND
         (p_cond_id   = st_rule.cond_id) THEN
        lv_cond := st_rule.condition;
        lv_exit := 'TRUE';
        EXIT;
      END IF;
    END LOOP;
  END LOOP;
EXIT WHEN lv_exit = 'TRUE';
END LOOP;
RETURN (lv_cond);
END Calculate_Commission;

FUNCTION Display_Valid_LOV (p_lov IN st_item.item_name%TYPE , p_form IN st_form.form_name%TYPE) RETURN VARCHAR2 IS
FUNCTION to determine is a LOV should be displayed
CURSOR cur_lov IS
SELECT COUNT(1)
FROM st_form f
, st_form_item fi
, st_item i
WHERE f.form_name = p_form
AND f.form_id = fi.form_id
AND fi.item_id = i.item_id
AND i.item_name = p_lov;
lv_count NUMBER := 0;
BEGIN
OPEN cur_lov;
FETCH cur_lov INTO lv_count;
CLOSE cur_lov;
IF lv_count > 0 THEN
RETURN (TRUE);
ELSE
RETURN (FALSE);
END IF;
END Display_Valid_LOV;

FUNCTION Get_Form_Size (p_value IN st_system_parameters.par_name%TYPE) RETURN NUMBER IS
FUNCTION to get the size of a form
CURSOR cur_attribute IS
SELECT par_value
FROM st_system_parameters
WHERE par_name = p_value;
lv_value st_system_parameters.par_value%TYPE := 0;
BEGIN
OPEN cur_attribute;
FETCH cur_attribute
INTO lv_value;
CLOSE cur_attribute;
RETURN (lv_value);
END Get_Form_Size;
FUNCTION Get_Cust_Grp (p_cust_id IN st_customer.cust_id%TYPE) RETURN NUMBER IS

/* Function to get the customers group */
CURSOR cust_grp IS
SELECT cust_grp_id
FROM st_customer
WHERE cust_id = p_cust_id;
lv_cust_id st_customer.cust_grp_id%TYPE;
BEGIN
OPEN cust_grp;
FETCH cust_grp INTO lv_cust_id;
CLOSE cust_grp;
--
RETURN lv_cust_id;
END Get_Cust_Grp;

FUNCTION Get_Sales_Grp (p_sales_id IN st_salesman.salesman_id%TYPE) RETURN NUMBER IS

/* Function to get the salesmans group */
CURSOR sales_grp IS
SELECT salesman_grp_id
FROM st_salesman
WHERE salesman_id = p_sales_id;
lv_sales_id st_salesman.salesman_grp_id%TYPE;
BEGIN
OPEN sales_grp;
FETCH sales_grp INTO lv_sales_id;
CLOSE sales_grp;
--
RETURN lv_sales_id;
END Get_Sales_Grp;

FUNCTION Get_Prod_Grp (p_prod_id IN st_product.prod_id%TYPE) RETURN NUMBER IS

/* Function to get the product group */
CURSOR prod_grp IS
SELECT prod_grp_id
FROM st_product
WHERE prod_id = p_prod_id;
lv_prod_id st_product.prod_grp_id%TYPE;
BEGIN
OPEN prod_grp;
FETCH prod_grp INTO lv_prod_id;
CLOSE prod_grp;
RETURN lv_prod_id;
END Get_Prod_Grp;

FUNCTION Get_Sequence_Number (p_seq_name INV ARCHAR2) RETURN NUMBER IS
    lv_string ARCHAR2(200);
    lv_cursor_handle INTEGER;
    lv_sequence NUMBER;
BEGIN
    lv_string := 'SELECT ||p_seq_name||.NEXTVAL FROM DUAL';
    lv_cursor_handle := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQL.PARSE (lv_cursor_handle, lv_string, 1);
    DBMS_SQL.DEFINE_COLUMN (lv_cursor_handle, 1, lv_sequence);
    IF DBMS_SQL.FETCH_ROWS (lv_cursor_handle) != 0 THEN
        RETURN (lv_sequence);
    END IF;
    --
    DBMS_SQL.CLOSE_CURSOR (lv_cursor_handle);
END Get_Sequence_Number;
END st_comm;