COPYRIGHT AND CITATION CONSIDERATIONS FOR THIS THESIS/ DISSERTATION

- **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

- **NonCommercial** — You may not use the material for commercial purposes.

- **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.

**How to cite this thesis**

INTEGRATION OF A
CASE TOOL AND A SOFTWARE ENGINEERING METHODOLOGY

by

MANFRED PAUL HERTENBERGER

A dissertation
submitted in fulfilment of the
requirements for the degree of

MAGISTER SCIENTIAE
in
COMPUTER SCIENCE
in the
FACULTY OF SCIENCE
at the
RAND AFRIKAANS UNIVERSITY

SUPERVISOR: PROF. J.H.P. ELOFF

MAY 1995
OPSOMMING

In hierdie verhandeling word die integrasie tussen CASE hulpmiddels en metodologieë ondersoek. Daar word veral aandag geskenk aan die objek-georiënteerde benadering om die integrasie probleem op te los. Die sogenaamde terugvoerlus word as 'n oplossing vir die integrasie probleem voorgestel en bespreek.

'n Aantal CASE hulpmiddels bestaan. Sulke hulpmiddels word deur stelselontwikkelaars gebruik om verskeie probleme op te los en om nuwe stelsels te skep. Daar kan ook van 'n wye verskeidenheid metodologieë gekies word. Tog is daar baie min metodologieë wat spesifiek gerig is op die gebruik van een sekere CASE hulpmiddel. Die verhandeling poog om die direkte integrasie tussen 'n CASE hulpmiddel en 'n metodologie voor te stel met die hulp van die terugvoerlus.

'n Aantal studies het beide CASE hulpmiddels en metodologieë ondersoek. Aan die begin van die verhandeling word die huidige tegnologieë bespreek. Probleme wat met tradisionele metodes ondervind is, word uitgelig.

Die terugvoerlus word op 'n stapsgewyse manier opgebou en bespreek. Die komponente wat gebruik word om die terugvoerlus op te bou word ook bespreek; veral die objek-georiënteerde benadering, wat as die beste boublok aanskou word. Deur die aanwending van die objek-georiënteerde benadering is dit moontlik om die vloei van informasie tussen die CASE hulpmiddel en die metodologie makliker en meer presies voor te stel. Daar word verder relasionele tabelle en gestoorde procedures aangewend.

Klem word geleë op die funksionaliteit van die terugvoerlus. Met ander woorde, moet die terugvoerlus aan sekere vereistes voldoen. Een van hierdie vereistes is die voorsiening van voldoende integrasiemeganismes. Data, proses en [control] integrasie word as baie belangrik geag en word bespreek. 'n Voorbeeld van hoe die terugvoerlus gebruik sou kon word, is ingesluit.

'n Kort opsomming sluit die verhandeling af.
This study addresses the topic of integration between CASE tools and software engineering methodologies. Specific attention is given to the object-oriented paradigm to solve the problem of integration. The component developed in this study to handle the integration is the so-called feedback loop mechanism. The feedback loop is the focus of the study.

A number of CASE tools are available today. These tools help developers achieve specific aims and to solve specific problems. Similarly, a number of different software development methodologies abound. Due to the fact that these methodologies are usually not specific to the use of CASE tools, these methodologies can be used for a variety of development environments. The problem addressed by this study is the direct integration of CASE tools and software engineering methodologies.

A number of studies have been carried out with regard to both CASE tools and software engineering methodologies. A discussion of current trends in both the CASE tool and methodology arenas is presented at the beginning of this study. Also, the problems experienced with more traditional approaches are discussed.

Given that the feedback loop mechanism is developed during the course of this study, a detailed discussion is presented regarding the components to be used in the construction of the feedback loop. The object-oriented paradigm is presented as the best solution to the integration problem. In fact, the object-oriented paradigm forms the core of the feedback loop. By making use of the object-oriented approach, the processes involved in the integration of the CASE tool and the methodology may be mapped accurately. Repository structures such as relational tables and stored procedures, or triggers, are also used in the construction of the feedback loop.
SUMMARY

During the course of the study, a number of demands are placed on the feedback loop. Of these, the tight integration demands are seen to be the most important. Such concepts as data, control and process integration are presented and discussed in detail, with specific regard to the operation of the feedback loop. A detailed example of how the feedback loop mechanism could be employed in a "real-world" situation is included.

A short summary of what was achieved during the course of the study is presented in conclusion. Those areas requiring more attention are mentioned.
ACKNOWLEDGEMENTS

This dissertation would not have been possible without the help of numerous people. Particularly, I would like to thank:

Prof. Elof for guiding me and lending a helping hand whenever required.

The CASE specialists at Siemens Nixdorf in Germany for their input.

My co-workers for their interest and support.

My father, mother and sister whose love and support made this possible and worthwhile.

Manfred Hertenberger
Johannesburg
May 1995
TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION .................................................................................. 1
1.1 Introduction .................................................................................................. 1
1.2 State of the art overview ............................................................................. 3
1.3 Defining the terminology ........................................................................... 4
1.4 The problem definition .............................................................................. 5
1.5 Overview of chapters ................................................................................. 6

CHAPTER 2: INTRODUCTION TO CASE ............................................................ 8
2.1 An introduction to CASE technology ......................................................... 8
2.2 Defining CASE ............................................................................................ 10
2.3 The workings of CASE ............................................................................. 18
2.4 The importance of CASE .......................................................................... 20
2.5 Types of CASE ........................................................................................... 21
2.6 Present day CASE ...................................................................................... 23
2.7 Future trends ............................................................................................... 24
2.8 CASE and the model .................................................................................. 26
2.9 Conclusion ................................................................................................... 27

CHAPTER 3: SOFTWARE ENGINEERING METHODOLOGIES ...................... 28
3.1 Introduction .................................................................................................. 28
3.2 Concepts of software engineering ............................................................... 28
3.3 Defining the software process .................................................................... 31
3.4 Software development processes ............................................................... 32
3.5 Software design issues ............................................................................... 36
3.6 Software specification ............................................................................... 38
3.7 Object-oriented design issues .................................................................... 39
3.8 Methodologies and the model ................................................................... 40
3.9 Conclusion ................................................................................................... 41

CHAPTER 4: THE PROBLEM DEFINITION ....................................................... 42
4.1 Introduction .................................................................................................. 42
4.2 Abstract ....................................................................................................... 42
4.3 The problem ................................................................................................ 43
4.4 Conclusion ................................................................................................... 46

CHAPTER 5: DERIVING A BASE FOR THE MODEL ...................................... 47
5.1 The problem definition ............................................................................... 47
5.2 The problem explained by way of an example ......................................... 47
5.3 A solution to the problem ......................................................................... 48
5.4 The feedback loop ..................................................................................... 48
5.4.1 The environment for the feedback loop .............................................. 48
# TABLE OF CONTENTS

5.5 The object-oriented approach ............................................................... 50
  5.5.1 The object .................................................................................. 52
  5.5.2 The object-oriented approach and information modelling .......... 53
  5.5.3 Advantages of the object-oriented approach ......................... 54
  5.5.4 The model and the object-oriented approach ....................... 55
  5.5.5 Information storage ................................................................. 56
  5.5.6 Constructing usable objects ..................................................... 57
  5.5.7 The object template ................................................................. 57
  5.5.8 Using the object template ......................................................... 60
  5.5.9 Inter-object communication .................................................... 61
  5.5.10 Introducing inheritance .......................................................... 64
  5.5.11 Concluding the object-oriented approach .......................... 67

5.6 Integration of diagrams .................................................................... 68
  5.6.1 Integration of diagrams through modelling transparency ........ 69
  5.6.2 The role of diagrams ................................................................. 70
  5.6.3 Modelling transparency ............................................................ 73
  5.6.4 Dependencies among diagrams .............................................. 74
  5.6.5 Sequential dependency ............................................................. 75
  5.6.6 Parallel dependency ................................................................. 77
  5.6.7 Degrees of modelling transparency ........................................ 78
  5.6.8 Degrees of modelling transparency and the feedback loop ...... 79
  5.6.9 The proposed model and the integration of diagrams .......... 80

5.7 Conclusion ......................................................................................... 82

CHAPTER 6: DEFINING THE CASE ENVIRONMENT ................................ 83
  6.1 Introduction ................................................................................ 83
  6.2 Defining the CASE tool component ........................................ 83
    6.2.1 Creation and modification of diagrams ............................... 84
    6.2.2 Deriving the diagram object class ..................................... 86
    6.2.3 Deriving a subclass for entities and attributes ................. 89
    6.2.4 Deriving a subclass for entity relationship diagrams ....... 92
    6.2.5 Deriving a subclass for data flow diagrams .................. 95
    6.2.6 The diagram class hierarchy ............................................ 100
  6.3 Conclusion .................................................................................... 101

CHAPTER 7: DEFINING THE METHODOLOGY ENVIRONMENT .......... 102
  7.1 Introduction ................................................................................ 102
  7.2 Defining the methodology component ..................................... 103
  7.3 Identifying the problem space .................................................... 104
    7.3.1 Identification of entities .................................................... 104
    7.3.2 Identification of attributes .............................................. 106
    7.3.3 Identification of relationships ....................................... 109
    7.3.4 Identification of data flows ............................................ 111
  7.4 Deriving the methodology object class .................................. 113
# TABLE OF CONTENTS

7.4.1 Deriving the project information system object class .................................. 114
7.4.2 Deriving the methodology object class ....................................................... 117
7.4.3 Deriving the entity object class ..................................................................... 120
7.4.4 Deriving the entity-attribute object class ..................................................... 122
7.4.5 Deriving the relationship object class ......................................................... 123
7.4.6 Deriving the data flow object class .............................................................. 125
7.4.7 The methodology class hierarchy ............................................................... 127
7.5 Conclusion ....................................................................................................... 127

CHAPTER 8: DEFINING REPOSITORY STRUCTURES .................................................. 129
8.1 Introduction ...................................................................................................... 129
8.2 Useful repository structures .............................................................................. 129
8.3 Extended relational systems ............................................................................ 130
8.3.1 Object data management ............................................................................ 130
8.4 Repository structures for the CASE and methodology components .............. 136
8.4.1 Repository structures for the CASE component ........................................... 136
8.4.2 Repository structures for the methodology component ............................... 142
8.5 Conclusion ....................................................................................................... 148

CHAPTER 9: THE FEEDBACK LOOP .................................................................... 149
9.1 Introduction ...................................................................................................... 149
9.2 An overview of the feedback loop .................................................................... 149
9.3 Defining the functions of the feedback loop .................................................... 150
9.3.1 Integration demands ................................................................................. 150
9.3.2 The integration framework ....................................................................... 154
9.3.3 Concepts of integration frameworks ......................................................... 154
9.4 The feedback loop as an integration framework .............................................. 161
9.4.1 Information exchange .............................................................................. 162
9.4.2 Maintain consistency and store global information .................................... 162
9.4.3 Integrating the CASE tool and the methodology ....................................... 164
9.5 Defining the feedback loop in terms of an integration framework ................. 165
9.6 Functional aspects of the model ..................................................................... 166
9.7 Creating the feedback loop ............................................................................. 167
9.7.1 Repository structures for the feedback loop ............................................. 167
9.7.2 Using the feedback loop control tables .................................................... 171
9.7.3 Interdependency within one component .................................................. 174
9.7.4 Revisiting diagram modelling transparency ............................................. 175
9.8 Defining the feedback loop class hierarchy .................................................... 176
9.9 Communicating change to the feedback loop ............................................... 191
9.10 Conclusion .................................................................................................... 197
LIST OF FIGURES

Figure 3.1 A general software life cycle .................................................34

Figure 4.1 CASE and methodology, standing apart.................................44
Figure 4.2 CASE tool enveloped by a methodology.................................45
Figure 4.3 Methodology enveloped by a CASE tool.................................45

Figure 5.1 Modelling transparency and diagram relationships....................72
Figure 5.2 Information sharing using a repository ..................................74

Figure 6.1 Relationships in an entity-relationship diagram.......................84
Figure 6.2 Attributes in an entity-attribute-relationship diagram ..............85
Figure 6.3 Data flow diagram .............................................................86
Figure 6.4 The diagram class hierarchy ...............................................100

Figure 7.1 SENSOR object example ......................................................105
Figure 7.2 OWNER object with attributes .............................................107
Figure 7.3 Adding a new object and moving attributes .............................107
Figure 7.4 Identifying redundant objects ..............................................108
Figure 7.5 Transferring attributes to the OWNER object .........................108
Figure 7.6 One-to-one relationship ......................................................109
Figure 7.7 One-to-many relationship ....................................................110
Figure 7.8 Many-to-many relationship ..................................................110
Figure 7.9 Parallel activities ...............................................................112
Figure 7.10 Display message flow .......................................................112
Figure 7.11 Save employee record data flow .........................................113
Figure 7.12 The methodology class hierarchy .......................................127

Figure 9.1 Integration possibilities .......................................................155
Figure 9.2 Construction of the model ...................................................166
Figure 9.3 Feedback loop operation .....................................................172
1.1 Introduction

It is an indisputable fact that the science of software engineering is currently undergoing a major change. On the one hand are the faster and more efficient hardware platforms that provide the power for large scale applications. On the other hand are the new and exciting methods that may be followed to create these applications. Object-oriented programming (OOP) and object-oriented design (OOD) are two examples.

Computer aided software engineering, or CASE for short, has been seen as a solution to the problem of producing good and robust software. Tools that help with regard to the design and sometimes generation of the new software are useful to programmers and designers. Such CASE tools range from simple program generators to sophisticated graphical tools. Prototyping facilities, screen painters and code generators are often an integral part of such tools; at other times these may be purchased as additional features to existing tools.

CASE tools alone cannot support the development cycle of new software. Such a tool is simply a way of creating the final product. Methodologies formalise the steps taken in the development cycle of new software. Such software engineering methods have been practised for a long time. The methodology creates a formalised flow according to which the new software product is to be created. Various methodologies have been created and used with varying degrees of success.

Possible criticism may be levelled at both CASE tools as well as software engineering methodologies as they exist today. CASE tools offer very little in the way of formalised software creation. Instead, such tools provide the facilities to produce the software using a particular programming language or database system. Software engineering methodologies offer a recipe according to which software can be produced. The steps to be followed in such a methodology are usually independent of any particular programming language or database system, for example.
Studies have been carried out in both the fields of computer aided software engineering, as well as software engineering methodologies. Very few studies have attempted to combine the two technologies in such a way, that each could gain from the other. In the end, such a relationship should result in a very useful tool for designers, programmers and analysts.

It is the aim of this study to combine CASE and software engineering methodologies. The result is a model that illustrates the benefits and practical advantages that may be gained from such a combination. Although a theoretical model cannot be proven to be correct until it has been tested in practice, real world problems are taken into account. The model serves to propose a new way of thinking in the CASE world with the aim of combining the traditional CASE tool with the traditional software engineering methodology.

It is quite clear from the literature that the object-oriented approach already plays an important part in CASE tools. There is currently no reason to believe that this will change. In fact, as the object-oriented technology evolves over time, it will play an even bigger part in all facets of computing - not only the CASE arena. It is for this reason that the proposed model draws heavily on the object-oriented approach.

A recent trend has been the movement towards the acceptance and general implementation of the graphical user interface (GUI). Sophisticated CASE tools have already made use of this paradigm. Graphical representation is generally considered to be one of the most effective methods for presenting complex information. The GUI furthermore has the advantage that novice users are able to become productive in a short time span. The object-oriented approach is ideal, but not necessary, for such a graphical presentation layer.

It is important to note that the model is an approach in preparation for later practical studies and implementations. The result of the study is the feedback loop, linking the two components by way of dynamic data exchange.
1.2 State of the art overview

The software tools used for the creation of new software are still in a process of evolution. [MACR87] Similarly, software engineering methods are still being developed. [MACR87] However, object-orientation is the technology that is replacing today’s database and programming technology for the design and development of computerised application systems - the technology that some regard as the ultimate paradigm for the modelling of information. [HARE94]

A number of CASE tools and CASE environments are currently available. Similarly, a number of software engineering methodologies abound. Of the tools available, a number of tools cater only for specific requirements. Other tools offer a wealth of features. A number of possibilities and products exist for developers to create so-called toolsets - environments in which all tools have the ability to communicate with each other as the development process is carried out. [DALE93] However, the integration of these tools is still regarded as being in its infancy. [PAGE93]

Integration demands made on tools depend on the integration demands made by the developer. In certain situations, complete integration may be desired. In other situations, no integration may be desired. To this end, integration frameworks must, and do, exist. [DALE93] Integration according to the chosen framework is always ideal. [DALE93]
Chapter 1: INTRODUCTION

1.3 Defining the terminology

In order to avoid confusion, the most important terminology as used in this study will be presented. These most important terms will be defined briefly.

CASE, or Computer Aided Software Engineering, refers to the process of creating and/or designing software systems by employing a software tool. The software tool provides the facilities and functionality to create and design a new software product.

Software engineering refers to the establishment and use of sound engineering principles and good management practice together with the evolution of applicable tools and methods to obtain software that is of high quality in an explicitly definable sense. [MACR87]

A software engineering methodology may thus be regarded as the normalisation of the sound engineering principles mentioned above. The methodology is a list of tasks in order of their priority. By following the steps proposed by the methodology, the designer or developer should be capable of producing software that is of a high quality in an explicitly defined sense.

The concept of object-oriented programming refers to a style of programming that provides ways of modularising programs by establishing partitioned memory areas for both data and procedures that can be used as templates for spawning copies of such modules on demand. [TELL89]

When referring to objects, a copy of such a module is meant. The object contains data and procedures and objects are capable of communication by means of message passing mechanisms. The concept of inheritance, by which
a new object may be created by using an existing object as a template, is an important benefit of the object-oriented approach.

1.4 The problem definition

The aim of this study is to provide a possible solution to the integration problem existing between a CASE tool and a software engineering methodology.

The problem that exists may be stated simply as follows: a number of software engineering methodologies exist. Also, a number of CASE tools exist. The methodology used by a developer is usually not specific to the CASE tool being used; conversely, the CASE tool does not take any particular methodology into account. The most important statement to be made from these facts is that the issue is not one of not having the tools, but rather how to use the tools in a consistent and integrated fashion. [DALE93] Stated simply, the problem definition states that software engineering methodologies and CASE tools stand apart.

Thus, the proposed model developed in this study will attempt to tackle the integration of a software engineering methodology with a tool for Computer Aided Software Engineering. It is the aim of this study to create an environment in which each of the two components, the CASE tool and the methodology, are able to exert the same influence on the developer. Both should be used in all aspects of the design and development process, and they should be co-operative in helping the developer reach the goal of creating a quality software product.
Chapter 1: INTRODUCTION

1.5 Overview of chapters

This chapter has served as a general introduction to the direction this study will be taking. The arrangement followed in the rest of this study is discussed briefly.

Chapter 2 provides an overview of CASE technology, the various CASE tools and toolsets available and explains CASE technology.

Chapter 3 provides an overview of software engineering methodologies and the various methodologies.

Chapter 4 explains the problem definition for this study in more detail.

Chapter 5 defines the environment for the study and introduces the technologies that will be used, such as object-orientation.

Chapter 6 defines the CASE component using the object-oriented paradigm as defined in chapter 5.

Chapter 7 defines the methodology component using the object-oriented paradigm as defined in chapter 5.

Chapter 8 gives a short overview of databases and repositories and describes the benefit of the repository approach. The repository structures for both the CASE component and the methodology component are defined.

Chapter 9 defines and illustrates the operation of the feedback loop in detail. The CASE and methodology components are used as presented in chapters 6 and 7. The table structures for these components are used as defined in chapter 8. Object definitions and table structures are defined for the feedback loop.
Chapter 1: INTRODUCTION

1.5 Overview of chapters

This chapter has served as a general introduction to the direction this study will be taking. The arrangement followed in the rest of this study is discussed briefly.

Chapter 2 provides an overview of CASE technology, the various CASE tools and toolsets available and explains CASE technology.

Chapter 3 provides an overview of software engineering methodologies and the various methodologies.

Chapter 4 explains the problem definition for this study in more detail.

Chapter 5 defines the environment for the study and introduces the technologies that will be used, such as object-orientation.

Chapter 6 defines the CASE component using the object-oriented paradigm as defined in chapter 5.

Chapter 7 defines the methodology component using the object-oriented paradigm as defined in chapter 5.

Chapter 8 gives a short overview of databases and repositories and describes the benefit of the repository approach. The repository structures for both the CASE component and the methodology component are defined.

Chapter 9 defines and illustrates the operation of the feedback loop in detail. The CASE and methodology components are used as presented in chapters 6 and 7. The table structures for these components are used as defined in chapter 8. Object definitions and table structures are defined for the feedback loop.
Chapter 10 presents the findings and concludes the study.
2.1 An introduction to CASE technology

The computer revolution has had a startling effect on the advancements made in both science and industry. Computers play an ever increasing role in very important tasks. These tasks reach as far as saving lives, defending nations, allowing global communication and exchanging currencies. [IEFT90] The importance of advancements in development tools to provide and support such tasks is clear.

The evolution of these tools has provided software developers with increasing flexibility and functionality. The first stage of this evolution was the advent of software engineering. [SODH91] By implementing engineering principles, computer scientists strived for more economical and reliable software. The debuggers and compilers that were used improved the speed and accuracy of software development. Methods such as structured analysis and design allowed the simplification of software development. Methods furthermore provided a generalised, rigid structure that could be followed for virtually any software development project.

Today, the need for software engineering and software development is becoming greater and greater as various applications are being found for automation. Large scale software development for a wide scope of applications is required more and more. The problem facing developers is the inability to create these systems fast enough. Errors or bugs in the software is a further problem.
CHAPTER 2: INTRODUCTION TO CASE

The most recent revolution has been the arrival of computer-aided software engineering. CASE has brought with it a number of improvements over previous methods of software development. One of these improvements is code and documentation generation. Front-end CASE tools exist which allow screens to be painted and objects to represent programming concepts. Back-end CASE tools are responsible for code and documentation generation. [IEFT90] The integration of both front-end and back-end CASE tools into one tool or environment is the most desirable result of CASE technology. Integrated CASE, or I-CASE, automates the entire system development cycle. In contrast, stand-alone CASE tools address only limited portions of this life cycle.

Various factors influence the development of software. Of these, the quality of the proposed system and its cost are perhaps the most important. Speed of development is important in cases where deadlines must be met. To overcome these problems, an integrated approach must be taken to the planning, development and construction of software. Such a process should take the above factors, as well as others into account.

Although CASE seems to solve many problems, there are certain new problems that an organisation must face once it starts using CASE technology. The most profound problem is the impact that CASE has on an organisation. The reason for this is that CASE requires an organisation-wide change. Changes must be made to switch from a traditional way of working to one that embraces CASE and uses CASE to its full potential.
2.2 Defining CASE

Computer an automatic electronic apparatus for making calculations or controlling operations that are expressible in numerical or logical terms [OXFO83]
Aided help [OXFO83]
Software programs for a computer [OXFO83]
Engineering application of science for the control and use of power [OXFO83]

The above definitions convey the message of what CASE is about. Thus, CASE is the application of certain methods to develop new software by making use of a computer with the aid of specialised existing software. The specialised existing software in this broad definition represents the actual CASE environment or CASE tool.

A definition that is more workable and certainly more general, given by [CASE86], is the following:

Computer-aided software engineering (CASE) is the application of automated technologies to the software engineering procedures.

This definition ensures that the entire range of software, from individual task-oriented tools to fully integrated operational software environments, is covered.

CASE environments are available for various applications and in various configurations. Their workings depend on the use they were designed for: rapid application design (RAD) tools allow the user to generate small applications very quickly. Program or code generators are similar in function to RAD tools. The
true importance of CASE environments however, lies in the integration of both these ideas. Integrated CASE environments satisfy both design and generation parts of the spectrum, whilst providing a multitude of other functions.

Although CASE environments are available on platforms of differing capability, most modern CASE environments have migrated to the world of the personal computer (PC). The graphical user interface and client server computing have made the PC a more accessible platform. The popularity of the "Windows" metaphor for operating systems has paved the way for easier to use software products. In this regard, a number of new products have appeared supporting this base. The point-and-click interface is sure to become the future of computing. With the advent of faster and more efficient processors being developed, the personal computer is moving into an arena that was previously the domain of mainframes and powerful mini computers only. The usage of CASE tools and environments can thus be seen to expand more rapidly and is available to a wider variety of users.

The general characteristics of CASE, according to [IEFT90], are the following:

a) breaking down of complexity
b) presentability to a wider variety of audiences
c) rapid prototyping
d) verifiability
e) increased maintainability
f) graphical user interface
g) allows intelligent project division and sub tasking
h) provides the power to produce the right system solutions faster
i) provides environmental independence
a) Breaking down of complexity

Both the design and coding of a software product are intrinsically complex. Even though this complexity is not as great for one project as it may be for another, problems are generally experienced during software design or coding. Design and coding are unfortunately not the only phases executed during the life cycle of a software product. Thus, added complexity due to other necessary phases arises. As stated by [TKAC94], CASE tools are often used to model the high-level enterprise and business areas, without integrating work from other design stages.

The aim of CASE environments is to minimise, if not to remove, these complexities. This is accomplished by providing the project team with integrated tools that encompass the entire software life cycle. The team can rely on a totally integrated approach to every phase of the software life cycle by making use of an automated CASE tool or environment. One of the best ways in which CASE tools are able to minimise complexity is by presenting everything to the user by means of a graphical user interface. [ADCY91] The user concentrates only on the most important elements, ignoring details that do not play a large role in the current stage of the software development project. In this way, these details are handled entirely by the tool.

b) Presentability to a wider variety of audiences

A software product must be created, evaluated and checked by a multitude of people. These people are both technically and non-technically oriented. The end user is usually not interested in the finer details of code that are required to create the finished product he will be using. Upper level management requires only certain information. The designer and programmer must have information concerning the entire project. [IEFT90]
CHAPTER 2: INTRODUCTION TO CASE

Representing the required information in such a way that each one of these people are able to understand it presents a problem. Software is seen as nothing else by an end user than as a collection of screens, forms or reports. Upper level management wishes to see graphs and pictures indicating how the software works. The designer requires flowcharts, entity-relationship diagrams and functional dependency diagrams. The programmer wishes to see source code and pseudo code.

A CASE tool is able to divulge information in any of these forms. [ADCY91] A database, also called a repository, holds all the information concerning the project. By setting filters and choosing various representation methods, anyone can examine the contents of the repository in a form that he will understand. [DATE90]

c) Rapid prototyping

Prototyping is used in all areas of production, from cars to packaging. It is only natural that the development of software should make use of this relatively inexpensive exercise. Prototyping allows a "dummy product" to be developed very quickly and inexpensively. This model or prototype is then reviewed. Once a review has taken place, the prototype is revised in terms of knowledge gained during the review. Then a new prototype is created, which is reviewed again. This cycle continues until the prototype reflects the facets of the proposed final product. [MULL90] states that prototyping is a "messy" process, however prototypes allow everyone to see the product in a nearly complete form. Making decisions based on written material and diagrams can never be as beneficial as allowing reaction with regard to a working prototype. [MULL90]

As was stated previously, the end user sees only screen layouts, forms and reports. All other details of a software product are hidden from view. Rapid
CHAPTER 2: INTRODUCTION TO CASE

prototyping in a software development environment thus takes the form of
creating screen layouts and presenting these to the user. The screens may
allow data entry but do no processing or calculation. The user can develop a
feel for the proposed system and offer suggestions that may make the final
system easier to use. [IEFT90]

Most CASE environments provide a rapid prototyping facility. The software
design team is able to walk through the painting of a screen. Once the screen
has been painted, the user may enter data into the fields. If the user is not
satisfied, changes can be easily made without any code ever being written. The
graphical user interface plays an important part in this facility. [MICR93]

d) Verifiability

No software ever written can be guaranteed to be 100 percent correct. It is
inevitable that errors or bugs will occur. [TURS87] This is entirely due to the
inability of a programmer to determine every possibility of execution a program
may allow. This problem is exacerbated as programs become larger and larger.
The more code is written, the more difficult it becomes to verify that the code
works correctly. Even if the code has been verified to be correct, other portions
of the code may influence the stability of the entire program or system. The
usual method of software verification was, and still is, peer checking. With this
method, every team member checks every program for correctness. Although
this is effective, human error can still result in bugs never being spotted. Unless
very rigorous verification takes place, programmers are often not very interested
in walking through someone else's code.

CASE tools attempt to separate the design process entirely from the code
generation process. In this regard, CASE tools attempt to verify the information
received from the developers against the actual user requirements. If the input data is incorrect, this verification has little hope of being error free. [GEHA86]

Furthermore, the user or developer has no need to see the code that represents the end product. Thus, the CASE tool alone is responsible for the generation of code and because almost no human intervention takes place, the automated tool can consistently produce error free code.

e) Increased maintainability

As is the case for software verification, the maintenance of an existing system forms an integral part of the software life cycle. As new requirements within an organisation are recognised, as more tasks are seen fit for automation and as external factors influence the working of a system, so the system must change to meet the new requirements. As most existing systems are poorly documented, making changes to such systems is not only very difficult but also error prone and expensive. [GILB88] Often a lot of time is wasted by programmers to understand the working of a particular program.

The repository of a CASE tool contains all the information pertaining to such a system. A CASE tool repository can be compared to a standard data dictionary in use by most modern relational database management systems (RDBMS). [DATE90] As a data dictionary in its own right, the repository stores all relevant data concerning the organisation and data concerning the software development projects undertaken by the organisation. The repository may be used to produce documentation, share data among members of the development team and act as a source of reference for future projects. All the data concerning the software is available on request for changes and additions. The maintenance of an existing system is thus no more difficult than the original design of the system.
As most CASE environments are graphically oriented, the system is easy to understand if icons or pictures are used. By making use of a pointing device such as a mouse, the user is able to make quick changes without having to worry about the impact the change may have on other elements of the system. The CASE environment takes care of all the changes, writes the new data to the repository and is ready to generate code for the updated software product. This ease of development makes for very easy and inexpensive maintenance.

**f) Graphical user interface**

The graphical user interface (GUI) has finally made its mark in almost every aspect of computer use. The success of the GUI can be attributed to the ease with which even novices are able to understand functions provided by programs. [MICR93] Objects employed in graphical environments such as push buttons and radio buttons are familiar to novices since these objects can be found in everyday life. Colour is used to highlight certain things and to draw attention to others. Icons or pictures are used instead of words or long descriptions to concisely illustrate functionality. Icons furthermore have the advantage of being internationally understandable. Instead of continuous and arduous keyboard operations the user can swiftly progress with a task by using a mouse or graphics tablet.

It comes as no surprise that modern CASE environments make extensive, if not exclusive, use of graphical interfaces. This fact has been illustrated more than once in the above discussions. All concepts can be represented by simple but effective pictures. This representation can be understood more readily by a wider audience than written documentation. [ADCY91] The use of liberal comments further extends the usability of the GUI; for a quick overview the graphical representation is sufficient. For a more detailed description the user may read the comments. Liberal use of colour in diagrams highlights various
functions or components. In this way, CASE environments are able to heighten productivity.

g) Allows intelligent project division and sub tasking

To maintain control over the development of a new software product requires strict project management principles and use of a methodology. [GILB88] The CASE environment supports these tasks by providing them to the project leader and other team members. Most CASE environments allow tasks to be subdivided amongst the team members. [ADCY91] [IEFT90] As one user designs screen layouts for one section of the system, another may design the accompanying database structures. In this way more work can be accomplished in less time. The CASE environment is responsible for the checking of integrity violations during such operations. [IEFT90] This feature is investigated in more detail as the model is developed in this study. Tight integration and control is one of the features of the model developed in this study.

h) Provides the power to produce the right system solutions faster

CASE environments have the ability to greatly enhance productivity. It has been found that the acceptance of a CASE environment as a very large part of an organisation is very difficult to achieve over a short time period. With proper training, education and motivation CASE can become an integral part of any organisation requiring software development. [IEFT90]

The GUI has the potential to reduce the learning curve often experienced by users of a new product. [MICR93] The integration of upper CASE for the design of a system, together with lower CASE for the generation of code is a further advantage. Due to the fact that the designers have more time to design the system rather than code it, the more one can be sure that the resulting product
will be a better product. [MULL90] The CASE environment can help in this respect by providing integrity checking, help and suggestions.

i) Provides environmental independence

The development of a new software system is a great expense to any organisation. CASE environments can help save both money and time by providing a further feature. This feature allows the reuse of existing modules for new systems. [CATT] Often, conceptual models may be used to generate code across different hardware platforms. In this way the organisation achieves environmental independence. This independence is very important for the organisation to cope with rapid changes in technology. [IEFT90]

2.3 The workings of CASE

The number of CASE tools available commercially is overwhelming. Also, CASE is not a fixed concept but rather a concept that is widely applicable. In this respect CASE has a very broad scope. In general, and according to the definition of CASE, a software tool is used to automate the software development process. This is in contrast with the conventional methods that implement paper and pencil technologies. According to [FISH], the ultimate goal of CASE technology is to separate the application program's design from the program's code implementation. Generally, the more detached the design process is from the actual code generation, the better the quality of the completed product should be. [FISH]

Such a tool is not the only aspect, however. To be implemented successfully, a CASE environment should consist of an automated tool and a methodology. The methodology is an important component of CASE for the following reason:
a methodology prescribes a way of work. The methodology provides a guide that can help to ensure success. A methodology is usually proven to hold certain benefits for those using it. In this way an organisation can be more reliant on the overall concept of CASE rather than on one particular automated tool. Traditional software development practices emphasise implementation, coding and debugging. In contrast, computer-aided software engineering focuses on requirements analysis and design specification. [FISH]

The workings of CASE or CASE environments differ according to both the methodology being used as well as the actual automated tool. The automated tool should tie into the methodology being employed. This methodology is a software development process or includes such a process. The methodology includes steps that must be followed to plan, develop and generate software. It is usual to think of such a development process as a cycle rather than as a linear process. In other words, the steps in the process are repeated as often as is necessary to develop and later to maintain the software. So called "feedback loops" may also exist. These loops allow certain steps to be iterated until a particular phase has been satisfactorily completed.

The strength of CASE lies in its ability to allow the entire development and maintenance cycle to be automated. The methodology forms a framework that encapsulates the CASE environment. Due to this automation, the steps taken are more precise and correct than those derived from more conventional processes. Complementary elements such as databases, programming languages and code generation modules may still be required for certain CASE tools. The trend of integrated CASE is certainly to provide an all encompassing tool or environment for software development.
2.4 The importance of CASE

Certain advantages of using CASE environments have already been mentioned. However, the importance of CASE is currently being noticed by more and more organisations intent on serious software development. Possibly the most important aspect of CASE is automation. Automation ensures that work is always done in a dependable way and frees people of boring and mundane tasks. In this respect modern CASE environments perform a very large portion of the work involved in the development of software.

Although the designer must still illustrate and describe the functions of the intended system, the CASE tool is able to assist by warning the designer of possible errors. The designer is usually required only to draw flow charts or draw diagrams that represent the functions the system is to perform. The CASE tool generates code which relieves the programmer of certain duties. Large software applications are usually critical to the success and orderly operation of commercial enterprises and to the reliable operation of national defence systems. [FISH] A study done in 1980 by TRW estimated that 64 percent of software errors arose from requirements analysis and design phases, and only 30 percent of these design errors were detected before the software was delivered for acceptance testing. [FISH]

Due to the automation of CASE environments scarce resources in the data processing department are no longer a major problem. The CASE tool is generally able to create program code. Thus, not as many programmers are required. The graphical nature of the CASE environment makes it easy to use. Furthermore, multiple tasks can be performed by one person simply because the time taken to develop the system is far shorter than usual. There is not as much reliance on specialists for the organisation which translates into cheaper software.
2.5 Types of CASE

The most important aspect of CASE, according to [SOMM92], is the methodology and strict tool integration. The integration of tool sets provides a more productive environment for the developers. [TKAC94] The methodology, or software development process, encapsulates the CASE tool. The methodology provides a guide for the development of software. The best known methodologies or similar concepts are the Application Development Cycle (AD/Cycle) in use by IBM and its associates, Method 1, Information Engineering and MERISE. [ADCY91] [QUAN91] Each of these development processes lists a number of steps that are repeated to develop a complete software product. The tools involved during this development are not important - rather, the function performed by these tools is important. [ADCY91]

The following is a list of various elements and tools involved in a CASE environment:

- cross life cycle tools: these are tools that are required during the entire software development life cycle. The tools include process management tools, project management tools and documentation preparation. [GILB88] The entire software life cycle is a process of maintenance, error correction and redevelopment. The management tools help the development team to maintain an overview of the life cycle. [HARM91]

- enterprise modelling tools: these tools facilitate planning and business modelling. The proposed system must fit into the organisation requesting it in such a way that productivity can be enhanced. Business modelling tools help with the identification of possible further automation and planning tools are used to plan the system. [GILB88]
• analysis and design tools: the analysis tools help to identify application requirements, data structures and the logic according to which the system should operate. [CASE86] The design tools allow screen design and prototyping. Prototyping has an important benefit in that it allows the end user to acquire a feel for the proposed system. [MULL90] This saves both time and money by allowing changes to be made "on the fly".

• programming languages: editors, compilers and debuggers to prepare and test the program code. If a code generator is used to create the complete system, these tools could be used by programmers to add in specialised code. [BOOC86] If no code generator is available, the team of programmers makes use of these tools to create the system by hand.

• generators: specialised tools performing such tasks as complete program code generation, specification generation and possibly generation of test data. The generation of code fragments can reduce development time drastically. [MULL90] Such code is usually error free, not requiring endless debugging sessions. Certain CASE environments can generate complete documentation. [IEFT90]

• test, maintenance and redevelopment tools: these tools help the development team with the testing of the system and parts of it, performing impact analysis and the maintenance of the system. The maintenance of an existing system is usually difficult. [TURS87] These tools help reduce this difficulty by storing data in repositories and integrating the necessary functions.

• knowledge-based systems: expert system technology helps with the enforcing of rules, specification of rules, reasoning and overall application creation. The knowledge base frees the designers from having to continually
recheck their own work. [ADCY91] User specific information can be stored and used by the expert system. Thus the CASE environment can be tailored for a particular environment.

- application development platform: services provided by workstations, library services, repository services and a methodology. The development platform forms the basis for the software development process. The methodology encapsulates the CASE environment.

2.6 Present day CASE

Software engineering is one of the most important aspects of the information industry. To provide organisations and users with high quality software is of the utmost importance. [HOAR75] The importance of high quality software is not to be underestimated.

Software engineering principles tackle the task of software creation using scientific, mathematical and engineering principles. The designer works according to structured methods that help ensure the high quality of the resulting software. The analysis and design phases of the method are the most important phases. They help the designer focus on the functions that the software is intended to perform. The better the understanding of the designers, the better the necessary functionality can be translated into requirements and specifications for the programmers.

Present day CASE environments have come a far way since the early days of software development. Such products as Excelerator and Bachman provide developers and designers with a rich set of features. [ADCY91] These products may be used during the complete life cycle of a software development process -
CHAPTER 2: INTRODUCTION TO CASE

from design to implementation. Maintenance of existing software is also supported to a large extent by these CASE environments. [ADCY91]

Although some CASE systems presume design and implementation methodologies, others provide more flexibility - and consequently less guidance in the way the work is to be done. [HUMP89]

2.7 Future trends

The time it has taken to create today's software development tools has been surprisingly short. CASE systems hold enormous potential if properly used, but they can also be costly mistakes. [HUMP89] A definitive future trend for CASE environments is the acceptance of this technology by more organisations and people within those organisations. Once the initial learning curve has been overcome a very powerful tool is at the disposal of the organisation. Ideally, one would never want to write a single line of code. Unfortunately, such a scenario is not very likely. Keeping up with the increasing demand for increasingly sophisticated software is virtually impossible. [FISH]

In the future, artificial intelligence systems and knowledge based systems are sure to make an impact. Already most CASE systems incorporate a knowledge base in some form or other. As this technology continues to evolve, expert systems should become even more of a help than they are currently. Intelligent CASE tools (ICT) will develop software for complex computer systems with a minimum of human interface. [SODH91]

Meta CASE systems are CASE systems that help developers create specific CASE systems. An organisation may thus have special requirements for all software that is required by that organisation. A specialised CASE system may
be too expensive. [BARK90] A cost-effective solution is the Meta CASE system. Such a tool generates a CASE system that falls within the required parameters. [SODH91]

The framework for bridging the automatic code generation gap is already in place. [FISH] Screen painters and screen designers have been in use for some time. A screen painter is not suitable for complex software development, however - these tools are usually only of use for simple database applications. The repository or data dictionary is fast becoming the most important aspect of CASE systems. The information in this repository pertains to the whole software development project. To become even more useful in the future, the repository will have to be able to include such information as data structure definitions and algorithms.

The integration of various tool sets is another important step in the evolution of CASE systems. As can be seen in some products available today, designers, diagram editors and code generators make up parts of such a tool set. The marketplace is sure to demand the capability to "plug" other design tools into the central design database. [FISH]

Future CASE environments should become more successful. CASE tools are currently not accepted by all developers, who do not believe that such a new technology can have a major impact on software development. This possibly stems from the fact that CASE is still in its infancy. Even so, it provides a number of advantages that previous software engineering methods lacked. Even more automation may hold the key to wider use of CASE tools. The more functions the tool can take over, the better. In this regard, the tool should relieve the developer of tedious tasks, allowing him to concentrate more on the important tasks involved in the design and development phase. Tight tool integration and integration of CASE tools and tailored methods is very important.
CHAPTER 2: INTRODUCTION TO CASE

2.8 CASE and the model

This study attempts to construct a model that incorporates a CASE tool component. It is important to focus on the aspects of "real-world" CASE tools that could impact the model and its operation.

Possibly the most important aspects of CASE environments that were mentioned in the preceding paragraphs are the concepts of verifiability, maintainability and project division and sub tasking. Why these three concepts were chosen will now be explained.

Verifiability does not include only the fact that a given program should function correctly. It is arguably more important that the program meets the user's requirements. [TURS87] Thus, it is important to map the user requirements to the actual code modules used to create the functionality for the user. It is usual that a number of design phases are examined before any code is produced. Once the designer has worked through the phases and stages of a chosen methodology, the results from these phases are passed on to a CASE tool environment. If the translation of information from the methodology to the CASE tool is not correct, the resulting product cannot be correct.

The concept of maintainability may be likened to that of verifiability, with the exception that maintainability usually deals with an already existing system. Maintenance of an existing system is usually successful only if sufficient documentation is available. Diagrams are a great help in dealing with the prospect of changing or enhancing a system. [GILB88] CASE environments provide such a function to ensure that the system can be changed more easily. Since changes to an existing system often require revisiting certain design phases, the CASE tool can often not be used in isolation; some integration is required from the CASE tool to the methodology that is used.
Intelligent sub tasking is usually possible only if the life cycle of a proposed software system has been planned in detail. CASE environments often include project management subsystems to cater for this requirement. [CASE86] The project plan would include all tasks that must be carried out to successfully complete the project. If a methodology is followed, all tasks in the methodology should be present in the project plan. This indicates a level of integration between a methodology and a CASE tool that is not immediately apparent.

The three paragraphs above have shortly illustrated benefits existing CASE environments can contribute to the model developed in this study. What is of particular importance for the model is the continuous requirement for integration between the CASE tool and the methodology. Methodologies will be discussed in more detail in the next chapter.

2.9 Conclusion

The previously mentioned points have only touched upon a very small area of the CASE subject. As can be seen, CASE is intrinsically complex, having to rely on a wide variety of components. The usage of CASE technology relies on the user having the correct training and knowledge to make effective use of a CASE environment. Currently, the move towards CASE tools by various organisations is increasing rapidly. Even though available CASE tools provide a rich set of features, it is very easy to lose sight of a goal by rushing into this new way of thinking. The most important aspect for the use of CASE technology is an open and positive approach.
3.1 Introduction

In the early years of programming, when FORTRAN was the language and ALGOL 60 a hope for the future, the software life cycle was almost a secret. Since then, various views concerning the software life cycle have been presented. According to [STEV91], there are two major reasons for using a software design method:

1. to aid in the design process
2. to communicate the result to others

Taking both these reasons into account when starting a new project can contribute to the success of the project.

3.2 Concepts of software engineering

[GILB88] states the disaster principle: disasters do not happen by accident; they are entirely creditable to our own management. Thus, management of any project plays a very important part in the software engineering cycle.

To further define the principles of software engineering, the explanation given by [GILB88] clearly defines software engineering. To begin with, software is all things which are not hardware in any computer system. Engineering is a process of design and trial construction of something, which aims to produce a system with a specified set of cost and quality attributes. According to [GILB88], software engineering is already so complex that a number of specialists are
CHAPTER 3: SOFTWARE ENGINEERING METHODOLOGIES

required to find the best designs. Whether or not this is so is a question of interpretation. In most cases the discipline of software engineering is well defined and precisely divided into tasks and subtasks. The fact that people make use of such a discipline causes problems in that the interpretation of the definitions and tasks may vary. Also, results of software engineering tasks must be communicated. This is possibly the biggest problem in software engineering. [GILB88] supports this by stating that communication between people is a huge problem in software engineering today.

A number of varying views on software engineering are offered by [LEDG87]:

- the software life cycle represents the cornerstone of software development
- the life cycle is out of date - it does not work with today's new technology
- get good people and the life cycle will manage itself
- prototypes are the answer, not the life cycle
- there are other ways to write software

Even though the above arguments may appear to be very different, the concept of a software life cycle is simply a matter of interpretation and definition. Even though certain actions taken by developers may not appear to resemble a life cycle, it usually true that development proceeds according to a number of distinct stages, where each stage is completed before the next begins. [LEDG87] This represents a life cycle - the more distinct and the more complete each stage is, the closer the life cycle is to being ideal. [LEDG87]

Software development is now seen as a crucial issue facing computer users in industry. The demand for more complex, high-quality software is growing at an unprecedented rate. There is always a need to engineer system software that will meet user requirements and expectations within available resources and that
can accommodate change throughout the software life cycle. [SODH91] The critical nature of the software development process is further highlighted by [SODH91]: the importance of software engineering is its systematic approach to software development, implementation, and maintenance throughout the life cycle of the computer system. [SODH91]

Requirements for modern day computer systems can further be identified as the availability and the consistency of the computer system. Consistent sub second response time and computer availability are critical to productivity. [JOHN89] To conform to these high standards, a new software system must be produced only after careful planning and thought. The direction a development chooses to take in order to reach their goal can be a critical success factor for the new software system.

Various terms and definitions are used to describe the process of the software life cycle and software design methodologies. Software engineering can be defined as the disciplined application of engineering, scientific and mathematical principles and methods in the economical production of quality software. [SODH91] The term information systems methodology is used to describe a methodical approach to information systems planning, analysis and design. [OLLE88]
3.3 Defining the software process

There are several reasons why the software process should be defined. The reasons given by [HUMP89] are:

1. to clearly define the roles of the involved people and organisations
2. to identify the routine tasks
3. to provide common direction for executing routine tasks, possibly through procedures or automation
4. to free the professionals from time-consuming routine and permit them to devote more energy to creative tasks
5. to provide a mechanism for continuously incorporating process improvements

[HUMP89] continues by stating that even though the need to control or eliminate routine through procedure or automation is obvious, the role of the methodology is not clear. The effective performance of complex tasks requires a clear understanding of human limitations. Because people can comprehend only limited amounts of detail, the common approach to complexity has been to establish hierarchies of abstractions to represent various levels of detail. [HUMP89]

As an example of this, the programs of a large-scale data processing system are thought of as the operating system, file management components, network management components and applications. If appropriate, each one of these components can be further subdivided. The same principles apply to the software life cycle or software development methodology: small tasks are used to break down a complex task into manageable components. As has already been stated, a life cycle is a scenario where the development process evolves in distinct stages, and each stage is completed before the next begins. [LEDG87]
While perfect life cycles may not exist, the life cycle is the goal or target for software development. [LEDG87]

It is with this in mind that the following section discusses a general life cycle.

3.4 Software development processes

Software design is a process of inventing and selecting programs that meet the objectives for a software system. [STEV91] Input includes an understanding of the following:

- requirements
- environmental constraints
- design criteria

The output of the design effort is composed of the following:

- an architecture design which shows how pieces are interrelated
- specifications for any new pieces
- definitions for any new data

An architecture depicts the relationships among the pieces that will comprise the software system. [STEV91] In this regard, diagrams are very useful to illustrate the architecture. Often, software design is a complicated task that involves balancing a number of constraints and criteria. Software design methods can assist by providing ways to describe evolving designs, as well as techniques and criteria for evaluating alternatives, all of which help to produce good designs. [STEV91] For the software design process to be successful, the designer must have a good understanding of the requirements, the hardware environment and
the software environment. According to [STEV91], this is a prerequisite for a certain design method to be selected. In other words, the requirements, hardware environment and software environment must be understood in order for an appropriate design method to be chosen. Thus, a useful order for design is not necessarily top down, but rather from independent to dependent requirements. [STEV91]

The essence of a life cycle has already been mentioned: distinct, complete stages, where one stage follows on the next once that stage is complete. [LEDG87] The life cycle is the goal or target of software development. In order to achieve this goal, every stage must be completed correctly and methodically. With a good understanding of the requirements, hardware environments and software environments, the designer stands a good chance of successfully completing the project.

Although professional software development will have different requirements for the software life cycle, a very simple life cycle can be used to illustrate the function of the life cycle. Such a life cycle could include the stages indicated in the following diagram:
This life cycle illustrates the most important concept of software life cycles, namely that every stage has a definite beginning and end, and that each stage treats a specifically defined set of issues. Software production, instead of being a continuing process of parallel, undiscriminated activities, is broken down into distinct stages. [LEDG87]

The above life cycle consists of six discrete stages.

1. **Project definition:** In this first stage, the emphasis is on outlining the proposed project, defining the resources needed for the project, negotiating with the user or customer and generally making a definite project plan.

2. **Formal specification:** This stage is more technical in nature. The exact specifications of the project are identified, the behaviour of the system is defined in detail and the protocols are established for the rest of the project. It is for this reason that this stage is often called the analysis phase. The major output of this stage is a formal specification, which is a technical
description of the entire project. An ideal output at this point is a preliminary user's guide to the proposed system. [LEDG87] Such a document has the effect of reminding the developer of all the requirements as the project progresses. It is of the utmost importance that any documentation produced should be approved and signed before any further development takes place.

3. design: critical data structures and algorithms are developed in the third phase. Furthermore, the general structure of the software is refined. [LEDG87] Prototypes may be built to allow experimentation with various possibilities and ideas. Thus, the viability of design ideas may be tested and discussed.

4. construction: in this stage, the final product is programmed, usually in a top down fashion. Integration of existing and new modules takes place to give the system shape and functionality. It is important that coding standards are adopted and implemented strictly. The output of this stage is a complete and working system.

5. testing: the testing stage involves the full use of the program in a live environment. It is here that both hardware and software are tested to the utmost to ensure that all anomalies are eliminated and corrected. [LEDG87] Testing of the new system should be as thorough and rigorous as possible.

6. installation: once the system has been completely tested and all errors have been corrected, the system is installed in a customer environment. Feedback from the customer and users is important to identify trouble spots that need to be redesigned or corrected. Reviews with customers place the developers in a position to identify all aspects of the system that are performing well, as well as those that are not working as expected.
The six steps mentioned above constitute the general life cycle. Each of the steps or stages is important to the overall software creation process. Except for the formal specification and design stages, all other stages are beyond the scope of this study. The design stage will be discussed shortly, followed by a discussion concerning the software specification process.

3.5 Software design issues

Software design encompasses a number of distinct activities. The three activities named by [FAIR85] are:

1. external design
2. architectural design
3. detailed design

External design deals mainly with the planning out of the proposed software product. In particular, the externally observable characteristics of the software product are specified. External design usually begins during the analysis phase and may continue into the design phase. The external, functional and performance requirements for the system are refined during the external design. In general, the view given by the external design phase is a high-level view of the proposed software product.

Internal design involves conceiving, planning out and specifying the internal structure and processing details of the software product. A result of the internal design phase is the specification of internal structure and processing details. The decisions are recorded to explain and motivate why certain concepts and
methods were chosen over others. A blueprint for the implementation, testing and maintenance of the software product is thus created.

During architectural design, the conceptual view of the software product is refined. Internal processing and structure details are refined and translated for use with the tool that will create the physical software product itself. These tools are programming languages, database management systems and CASE tools. Algorithms to provide the engine for the software product are also defined at this stage.

The concept of procedural abstraction is present in and during the architectural design phase mentioned above. As stated by [GEHA86], procedural abstraction can be viewed as a mapping from its inputs to its outputs. The specification of a procedural abstraction has two parts, namely the interface specification and the behavioural specification. Thus, the specification of algorithms and internal structures is a result of procedural abstraction.

An important point made by [FAIR85] is the use of abstraction during the software design process. Abstraction plays an important role in that it allows the designer or developer to organise and channel thought processes by postponing structural as well as detailed algorithmic considerations. [FAIR85] The concept of abstraction is used in the development of the model for this study. In particular, the concept of abstraction will be used to explain the importance of object-orientation for the proposed model. It is important to note that the concept of procedural abstraction mentioned above is different to the concept of abstraction defined here. Procedural abstraction helps to define physical structures. Abstraction is a process by which these physical structures are viewed in a simpler way.
3.6 Software specification

The concept of software specification is very important for the software engineer. As stated by [TURS87], one thing on which all software experts agree on, is the necessity to establish the correctness of every program and system that is delivered to customers or intended to be used otherwise.

Determining the attributes of quality in any software product is made difficult by the number of quality attributes that may be present. As an example, a certain program may be very efficient at performing some task, another program may be portable across different hardware platforms and yet another system may allow changes to be made very easily. Determining whether or not a software product measures up to such quality standards is open to interpretation. The most important aspect is that the software product does what it should in the way that it should be done, and that these operations are performed correctly. A program that is highly portable is useless if it does not work correctly. Thus, correctness is good, lack of it is bad. [TURS87]

Even though the topic of program correctness is beyond the scope of this study, it is important to realise how lack of correctness influences the design and development process of a software product. The software life cycle makes assumptions in that the initial conceptual design specification is used in further phases where this specification is refined and applied. It is thus reasonable to assume that an error or omission in the conceptual design phase will almost certainly be carried through to the final, detailed specification and the physical software product itself. Furthermore, given the fact that abstraction is used in these initial design phases, whereas structural definitions and algorithms are used in later phases should indicate that errors and omissions should be picked earlier rather than later. This is supported by [TURS87] who states that it is virtually impossible to prove that any given piece of program code is correct or
performs its task correctly. It is usually assumed that the code that performs its
task correctly, according to the design definitions, and handles out of bound
values, is correct. In other words, finding an error in a conceptual design view
should be easier than finding an error in a detailed design view, since the
detailed design view must take internal structures and algorithms into account.

3.7 Object-oriented design issues

The object-oriented paradigm is used extensively in this study to develop a
model for the integration of CASE tools and software engineering
methodologies. It is for this reason that the concepts of object-oriented software
design are discussed briefly.

As with any other information-oriented design methodology, object-oriented
design (OOD) creates a representation of the real-world problem domain.
[PRES87] The term problem domain is called the problem space by [COAD90]
and it is this term that will be used throughout this study. The unique nature of
object-oriented design is its unique ability to build upon three important software
design concepts, namely abstraction, information hiding and modularity.

The concept of abstraction was mentioned in a previous section. When a
software system is designed using the information hiding approach, each module
within the system hides the internal details of its processing activities and
modules communicate only through well-defined interfaces. [FAIR85] A major
benefit of information hiding is the ability to modularise the software system. The
third benefit of OOD is modularity. Information hiding aids the developer to
create modular software. Even though many correct, yet different, definitions for
the term module exist, a modular system is one in which various unrelated
components solve local aspects of a particular problem.
CHAPTER 3: SOFTWARE ENGINEERING METHODOLOGIES

The object-oriented paradigm is discussed in detail in later chapters. For now, it is important to be aware of the benefits of object-oriented design as mentioned here.

3.8 Methodologies and the model

This study attempts to construct a model that incorporates a methodology component. It is important to focus on the aspects of a "real-world" methodology that could impact the model and its operation.

Possibly the most important aspect of methodologies that was mentioned in the preceding paragraphs is the concept of a precisely defined set of steps that must be adhered to for the project to be successful.

The general life cycle model discussed earlier gave an idea of how the results of one stage are integrated into the next stage. The flow of information from one stage of the methodology to the next is important. Even more important is the fact that an error in an earlier stage directly affects the results of any other stage later on in the methodology. To correct an error made in an earlier stage may be difficult or easy depending on when the error was made. It is usually inconvenient at best to retrace steps that have already been followed and carried out.

In situations where the results of a methodology are fed into an automated tool, such as a CASE tool, making changes may be even more difficult. A channel of communication must be present to inform the CASE tool whenever a change has been made, or a new fact within the problem space has been recognised. The model developed in this study addresses this problem.
3.9 Conclusion

Software engineering is a discipline which has been in use for a long time and in different forms. The importance of methodologies is not to be underestimated. The guidelines set by any methodology ensure that the overall software life cycle follows the correct path.
CHAPTER 4: THE PROBLEM DEFINITION

CHAPTER 4
THE PROBLEM DEFINITION

4.1 Introduction

The introductory chapter has stated the problem this study is attempting to address. To make the problem definition clearer and complete, this chapter illustrates the problem in more detail. Chapter 2 and chapter 3 have discussed CASE environments and methodologies. The topic of discussion in those chapters, namely the CASE tool and the software engineering methodology, are the components which form a part of the model developed in this study.

This chapter serves as a precursor to the next chapter which deals with the derivation of the model and ultimately the feedback loop.

4.2 Abstract

The proposed model will attempt to tackle the integration of a software engineering methodology with a tool for Computer Aided Software Engineering. The direct feedback and communication between these two components of the integration is of the utmost importance.

The feedback loop has been mentioned briefly in the introduction. The environment necessary for the feedback loop to work consists of a CASE tool component and a methodology component. An overview of these components was given earlier.
4.3 The problem

Currently, a number of CASE tools are available. These range from automated tools that are specific to certain application areas, to systems that can help with the development of various computer software. Usually, the lifetime of the projects versus the lifetime of the tools creates problems. It is common that a project outlives a certain tool. If integration exists, the transition to a new tool should be less complicated than if no integration exists. [DALE93]

The characteristics that were mentioned to have an impact on the model developed in this study were discussed in chapter 2. In that chapter, it was shown that both maintainability and verifiability had a similar impact on the design and development of software when making use of a CASE tool. Sub tasking was illustrated as being an important integration factor by combining project information from a project management subsystem within the CASE tool and the stages and phases from a chosen methodology.

A number of different software development methodologies also abound. Due to the fact that these methodologies are usually not specific to the use of CASE tools, these methodologies can be used for a variety of development environments. The specifics of how to follow the phases the methodology presents are not important. Added to this comes the fact that software engineering products are becoming ever more complex, partly through the application of software to more technically difficult problems and partly through the sheer size of modern software projects. [MITC90]

Characteristics of existing methodologies are able to impact the model developed in this study. In particular, communicating changes made in one stage of the methodology to tools used in the design and development task can be viewed as a complex task. No matter how small the change that has to be made may be, the impact on the information gathered with the aid of the methodology is serious.

The main problem definition may thus be stated as follows: a developer may choose to follow the guidelines a particular methodology presents. The developer may choose to implement the actual software project using virtually
any tool that is available. The developer is not forced to use a CASE tool. Similarly, a developer may choose to make use of a particular tool for the implementation of a software project. There is no reason for the developer to even consider using a methodology. Even if a methodology is chosen by the developer, the methodology may make certain assumptions concerning toolsets used for production of software.

This is supported by [DALE93] who states that the number of tools which may be of benefit on a specific project is overwhelming, with estimates of over 1500 tools being available. The most important point made by [DALE93] is that the issues faced are not those of not having the tools, instead they are how to use the tools in a consistent and applicable fashion.

Thus, it can be stated that for most intents and purposes the CASE tool and the software engineering methodology stand apart.

![Diagram: CASE and methodology, standing apart](image)

As stated by [MITC90], software engineering methods are formalisms for describing software. The capabilities of tools are determined by the methods they implement. Furthermore, integrated development environments do not require rigid procedures for software development. [MITC90]

Two alternative ways of viewing the current disparity between the concepts CASE tool and methodology exist. In the first case, a CASE tool may be enveloped by a methodology. Here, the methodology serves as the driving force
behind the development. The developer acts upon phases and tasks prescribed by the methodology. The CASE tool is simply a vehicle to further the design of the software.

The second viewpoint views the methodology as being the vehicle to further the development of software. Here, a CASE tool is the dominant component, whereby the methodology is simply used as a point of reference.

It is the aim of the proposed model to bring these two concepts into a single environment. In other words, the aim of this study is to create an environment in which each component is able to exert the same influence. It should no longer
be the CASE tool that drives the development of a software product, with a software engineering methodology in tow. In the same way, a software engineering methodology should not be able to govern the design and development of a software product with the CASE tool being an aside. Both components are to have input into all aspects of the development and design. The feedback loop is the mechanism to accomplish this.

4.4 Conclusion

In this short chapter the problem definition was explained. The following chapters define the components required for the problem to be solved.
5.1 The problem definition

As was stated in a previous chapter, the problem this study is attempting to address is the following: to tie information from a CASE tool or environment together with information gained from some software development methodology. It is the aim of this study to develop a mechanism to achieve this integration.

Both the CASE tool and the methodology will often be referred to as components. For purposes of this study, the integration or bonding mechanism between the components will be known as the feedback loop.

5.2 The problem explained by way of an example

A number of integration possibilities exist between a CASE tool and a methodology. Thus, a developer decides to follow a certain methodology. In order to make the design process of the required software system easier, he or she decides to make use of a commercially available CASE tool. The design process starts off with the developer creating a project plan that covers all aspects of the design and development process. If the methodology requires a data flow diagram to be created, the developer initialises or switches to the CASE tool and creates the DFD. Once the DFD has been drawn up and all its components defined, the developer starts up the project management package and alters the project plan to reflect the progression of his or her work.

The problem with this approach is simply that the developer has the responsibility of ensuring that all steps are carried out. If a step is ignored, there is no way the developer can be warned. The reason is that two distinct entities are being employed to carry out the same job, without any controlling mechanism being used.
CHAPTER 5: DERIVING A BASE FOR THE MODEL

5.3 A solution to the problem

The whole scenario changes if both the components are integrated. Thus, the developer once again selects a methodology and a CASE tool. Before starting on the design and development process, the developer instructs the integration mechanism of certain default information. This integration mechanism is the feedback loop. If the methodology requires a DFD to be created, the CASE tool will be informed of this fact. The CASE tool can then warn the user that a certain step must be completed before he or she can continue. In the same vein, the methodology component is continually aware of the steps that have been carried out and those which must still be carried out or need to be completed.

In this way, the feedback loop aids the developer, but also ensures that the entire design and development cycle is in a state of consistency.

5.4 The feedback loop

In order to define the feedback loop, an underlying structure must be created to support it. This simply means that the environment in which the feedback loop is to function must be defined. Furthermore, the components making up the environment must be described.

Thus, to define the functionality and the mechanisms of the feedback loop, a model must be created. This model has the function of providing the working platform for the feedback loop - in other words, an ideal environment that allows the feedback loop to be described.

5.4.1 The environment for the feedback loop

The model can be easily translated into real-world terms. It consists of two components, namely a CASE tool and a software design methodology. A choice would be made by the user of these components as to which specific CASE tool and methodology would be employed. The choice is governed only by the fact that both must be available in a computerised or automated form. For modern
day CASE tools this is not an issue. All are available as computerised aids to the developer. Methodologies are not usually available as automated tools. Transforming a methodology into an "automated" tool is usually very easy. By simply using a project management tool or even a spreadsheet program to map the steps and phases of the methodology, an automated form of a methodology can be created.

It is important that some form of communication is possible between the components. This can be established by a so-called "hot-link" when working in modern GUI environments, for example. The communication path is important as it forms the basis for the feedback loop. If no "hot-link" is available, an information import function is usually present. It should be clear that the type of link that is established between the components governs the efficiency of the feedback loop. The communication channel between the CASE tool and the methodology component is of the utmost importance - the entire operation of the feedback loop depends on some communication channel being present. The communication channel should not only be present, but should provide the functionality the feedback loop requires. Such functionality is the efficient and error-free transport of information in a form understandable to the feedback loop.

To develop an ideal model environment, two approaches will be pursued:

1. the object-oriented approach
2. the integration of diagrams through modelling transparency

The two approaches have been chosen for one reason, namely that they are suited to the environment in which the feedback loop is to operate. The object-oriented approach is widely regarded as an approach that has significant potential for influencing the field of software engineering. [LANO94] A lot of potential may be gained from the areas of re-use, software maintenance, compartmentalisation of system development and the scaling up of formal specification techniques to large systems. According to [WINB90], software reuse mechanisms such as inheritance make consistency easier to maintain. In the same vein, [PINS88] illustrates that the most important feature of any software development is the design of new structures of classes which make maintenance easy. These same advantages are applicable to the feedback loop, which will be shown to be an object-oriented system in its own right.
The integration of diagrams is an important addition to the object-oriented approach. Diagrams prove useful for abstracting complex issues and localising the problem areas. [SØLV93] The principle of abstraction may be defined as the suppression of irrelevant details in order to concentrate on the more important issues at various modelling stages. [SØLV93] The principle of localisation enables the designer to model individual properties of application objects separately and then integrate these descriptions to produce a complete design. [SØLV93]

It will become clear that the object-oriented approach is the main building block of the model. The reason for this has been mentioned above. The integration of diagrams is seen as an important addition to this approach, however, since diagrams are often found to be a focus of modern CASE tools. Thus, the approach focusing on diagram integration is discussed with object-orientation as a base.

5.5 The object-oriented approach

The object-oriented approach is to be a base for the proposed model to draw upon. [LANO94]'s argument is that the object-oriented approach has significant potential for influencing the field of software engineering, especially in the areas of re-use and software maintenance. Another reason is that the object-oriented approach is more geared towards the way humans think. [PINS88] The object-oriented approach has no limits on the available data structures or instruction sets for use in the problem solution. For this reason, there is no need to force the steps in a solution to fit around a limited set of constructs. [PINS88] It is this fact that makes the object-oriented approach so appealing and useful.
CHAPTER 5: DERIVING A BASE FOR THE MODEL

According to [COAD90], four factors have been key in promoting the use of object-oriented techniques:

1. The concepts surrounding object-orientation have had over one decade to mature. In fact, object-oriented systems have been in use for almost fifteen years. [WINB90] In this time, the basis for object-orientation has progressed to become a very powerful paradigm for the programming, design and analysis of software systems. [COAD90] This study is concerned with the design and development of complex software systems. It is for this reason that the object-oriented paradigm is of such importance.

2. The underlying technology for building systems has become much more powerful. Given time, object-orientation has become more visible as an efficient and cost-effective method of developing and designing software. [COAD90] This is mainly due to the concept of inheritance. Inheritance allows the construction of new objects by adding to existing objects. It will be seen that of all the properties of object-orientation, the notion of inheritance is possibly the most important.

3. Modern systems are very different from earlier systems. In every respect, today's systems are more complex, larger and continuously changing. Object-oriented techniques have proven themselves to enhance the ease of maintaining complex systems, as well as forming a solid base for creating new systems. [COAD90] The environment being described in this chapter is required to deal with changes in software systems. The feedback loop must communicate these changes from the CASE tool component to the methodology component and vice versa. Thus, the object-oriented approach must form a building block of the feedback loop and the described environment, in order for this function to be fulfilled effectively.

4. Modelling data has become more important than it ever was. Functional complexity is less important - organisations need access to large volumes of data and need to manipulate that data. Object-orientation has proven capable of handling these requirements. [COAD90] This is mainly by way of the modularisation and localisation described by [LANO94]. The feedback loop itself must contend with
data streams that are transferred from one component to the other. Keeping information in both components synchronised is very important and can be achieved by way of the message passing ability of objects.

A number of the advantages of the object-oriented approach are to be included in the design of the proposed model. Concepts such as encapsulation, inheritance and communication with messages are a few examples. Each of these concepts provides the ability for the object-oriented approach to be very powerful. These concepts will be illustrated later on.

5.5.1 The object

The term *object* has its roots in two diverse fields. Firstly, from information modelling where an object is a representation of a real-world thing and a number of instances of that thing. [COAD90] Secondly, from object-oriented programming languages where an object is a run-time instance of some process and values. In this case the instance is defined by a static description called a *class*. [COAD90] A *class* can be defined as a definition or template. In programming languages, such a definition would be equivalent to an instance variable of any particular type.

[COAD90] gives the following equation for recognising an object-oriented approach:

\[
\text{Object-oriented} = \text{Objects} + \text{Classification} + \text{Inheritance} + \text{Communication with messages}
\]

An object is an entity that has particular attributes, or data, and ways of behaving, or procedures. [WINB90] The way of behaving, or procedure, is formally known as a method.
A class, or classification structure, is a description of a set of nearly identical objects. [WINB90] A class consists of methods and data that summarise common characteristics of a set of objects. A subclass is a derived class. In other words, the subclass is simply a part of the class hierarchy.

Inheritance is the mechanism for automatically sharing methods and data among classes, subclasses and objects. [WINB90] Thus, new objects may be created simply by adding data or methods to existing objects.

Messages are required in an object-oriented system to get objects to become active. Action occurs whenever an object receives a message; that is, a request asking the object to behave in some way. [WINB90] Whenever object-oriented systems execute, there are messages being executed, received and sent by the objects within the system.

The use of objects in the proposed model will follow the equation given by [COAD90]. Thus, all objects used for the proposed model will provide the functionality and will be structured according to this equation. Even though this equation is very general, it does address the most important aspects of the object-oriented approach. More importantly, the equation allows the model to be described in well-defined terms, without the need to introduce too many new ones.

5.5.2 The object-oriented approach and information modelling

Information modelling has evolved over a number of years. The primary tool of information modelling is the entity-relationship diagram. When using information modelling to describe a system, objects in the real world are identified together with services for these objects and relationships among the objects. Services are the processing requirements for each object. [COAD90] This can be likened to the concept of messaging given by [WINB90]. By tying the concepts of information modelling together with the concepts gained from object-oriented programming languages, the object-oriented approach is formed.
CHAPTER 5: DERIVING A BASE FOR THE MODEL

Information modelling gives us the concepts of attributes, relationships, structure and the representation of an object as some number of instances of something in the problem space. Object-oriented programming languages give us the encapsulation of attributes and services as an intrinsic whole. [COAD90] Object-oriented programming languages deal with the technical nature of object use, namely the definition of classes, inheritance and communication via messages. Information modelling deals with the analysis of problem solving, by investigating the problem and creating a solution for it.

5.5.3 Advantages of the object-oriented approach

A number of advantages exist for the use of the object-oriented approach. These advantages are also important for the working of the feedback loop, in other words for its environment, as well as for the feedback loop mechanism.

The object-oriented approach defines and communicates requirements in a natural manner. Objects in the real world can be directly represented as objects in the object-oriented sense. A real-world object has attributes and actions performed on it. In the same way, an object in the object-oriented sense has attributes and services that work on the attributes. Thus, the translation proceeds automatically. This is especially important for the environment being defined in this chapter. Here, two components are described. In the end, both components must be integrated by way of the feedback loop. Creating these components in an object-oriented fashion provides the ability of these components to communicate by way of messages. The feedback loop is required since the CASE component and the methodology component are not created from the same building blocks and can thus not “understand” each other. Also, the CASE component would store different objects to the methodology component, for example.

The object-oriented approach treats attributes and services on those attributes as a whole. This concept is called encapsulation of attributes and services and is enforced by most object-oriented programming languages. As [MAND92] notes, encapsulation ensures that objects are self-contained and useful by
themselves. It requires that an object contains attributes and services that are known to that object. The attributes and services make up the object - thus the attributes and services represent the object. The methodology component could make use of this fact to store one phase of the methodology, as well as all its subtasks, as methods. The requirements and information necessary for that phase could be stored as the data component of this object. The entire methodology component could be a class that is made up of multiple subclasses similar to the one just described. In a similar way, the CASE tool component could store diagrams as objects. The entities and their respective attributes in the diagram could make up the data component of the object. The methods could be relationships to other diagrams that require information from this one. The object-oriented paradigm allows the creation of such objects with little difficulty.

A consistent and clear approach to problem solution is applied by the object-oriented approach. Real-world objects can be translated to objects in a more abstract sense. The structure of these objects is always the same, and is very simple. By investigating the object definition, the problem solution becomes clear. All objects in the proposed model can be derived from a single base model. This is true for both the CASE tool component as well as the methodology component. In this way, the modelling and mapping from one component to the other within the environment is simplified. This approach also makes the construction and operation of the feedback loop less complex.

5.5.4 The model and the object-oriented approach

In order to make use of the object-oriented approach for the model developed in this study, the object-oriented approach as it will be used must be explained.

Any object-oriented system, such as the proposed theoretical model, is based on objects as underlying building blocks. In the first instance, these building blocks may be used to create the system or application itself. In this case, the base software may have been written using an object-oriented programming language. It is useful to view the proposed model as a possible software implementation, since the model itself must incorporate the functionality of a
CHAPTER 5: DERIVING A BASE FOR THE MODEL

CASE environment with the scope of a methodology. In the second instance, the building blocks are used to store user-defined data while the software or application is in use. In this case, the application uses object classes to store user input. It will be illustrated later how the theoretical model addresses fulfils both criteria. The proposed model is to be made up of a number of objects. Also, the input gathered from the user will be stored as one or more objects.

In the proposed theoretical model, the implementation is less important than the way the model functions. A practical implementation of the proposed model's approach could be a software application. Such a software package could be written using a high-level object-oriented programming language, such as C++. This is not essential - it makes sense from current trends in the computer industry to move towards the object-oriented paradigm when coding software. Furthermore, such an approach makes the implementation of the model's functions a lot easier.

The most important function the model must provide is the ability to communicate information from the CASE component to the methodology component, and vice versa. This has been mentioned in the introductory paragraph. This problem will be illustrated in greater detail in the next chapter.

The technical issues of implementing the model as a software package will not be addressed in this study. As was mentioned before, however, it is useful and often necessary to view the proposed feedback loop as a software application that can interface the CASE and methodology environments seamlessly.

5.5.5 Information storage

Both components of the proposed model will store user data as objects. The user data referred to will mainly be data entered by the user. Since graphical user interfaces have been mentioned, input of information by the user could be in the form of diagrams. The user would be prompted, or could choose, to enter the required information. Diagrams could be generated from this information. By making use of the mouse or a graphics tablet, the user could further manipulate the diagrams to include relationships and entities, for example. According to
[COHE92] [MICR93], a good user interface is widely recognised to have a major impact on the use of software systems. Once again it must be stressed that the proposed model is not concerned with the implementation of a software package. The user interaction with such a system must not be disregarded, however.

The diagram approach couples directly to the information modelling approach once one views the diagram as a representation of a real-world thing or entity. This ties in with the definition given by [COAD90] and which was mentioned at the beginning of this chapter. Entity-relationship diagrams and functional dependency diagrams are two examples. The object containing the diagram data is an encapsulation of information and a description of its manipulation. The object maintains the data using corresponding methods to perform various actions. The methods are equivalent to the services mentioned earlier. Since the diagram is a representation of the object, the diagram can be seen as being more than simply an illustration. Instead, one can view the diagram as being composed of a number of objects, each being based on an object within the real world. It can be said that the object has data inside, and knows what to do with that data. [COAD90]

5.5.6 Constructing usable objects

To be able to create usable objects for the proposed model, a short discussion on practical object-orientation will follow.

5.5.7 The object template

The structure of some arbitrary object is given by the following template:

\[
\text{Object } \text{<object\_name>}
\]
\[
\{\text{<attribute\_list>}\}
\]
\[
\{\text{<method\_list>}\}
\]
Please note that both the `<attribute_list>` and the `<method_list>` are optional; the `<object_name>` is not. To make the naming conventions clear, the following scheme will be used:

- object names will always follow the format `somename_Object`
- attribute names will always follow the format `somename_Attribute`
- method names will always follow the format `somename_Method`

The above naming conventions should not be considered as restrictive; naming any of these three items is entirely arbitrary. The convention is simply introduced to prevent confusion and to enforce consistency.

The template just given is admittedly very general. It is useful for a number of reasons, however. The template contains everything that a general object should contain. Due to its general structure, it is useful in any situation, since all that is required is to flesh out the basic skeleton.

The object is given a distinct and unique identifier, the `object_name`. This name is used whenever the object is to perform some action. Specifically, the `object_name` will be used whenever one of the object's methods will be called.

The `attribute_list` is a list of properties that the object contains. The `attributes` are what the object 'knows'. An attribute is defined by a name, the `attribute_name`, and a type. The type details what the attribute is. If the type of the attribute is a string, for example, then that attribute will contain a sequence of alpha-numeric data. The attributes are data storage areas within the object.

Data types may be simple or complex. Simple data types are such data types as strings and integers. Complex data types are data types that are in themselves structures. Such data structures are arrays, linked lists or records - possibly even objects.

The definition for each entry in the `attribute_list` is expanded to look as follows:

```
somename_Attribute of type Type
```
CHAPTER 5: DERIVING A BASE FOR THE MODEL

The name may be chosen as desired. As a matter of syntax, the two words "of type" must appear in the attribute definition. *Type* denotes some data type, for example *Integer* or *String*. As an example, the following defines *Name_Attribute* as a data element of type *String*:

\[
\text{Name\_Attribute\ of\ type\ String}
\]

Complex data types must be specified together with their definition. In this case, the following syntax is used:

\[
\text{Datatype <type\_name>}
\begin{array}{c}
\text{} \\
\{\text{structure\_list}\}
\end{array}
\]

Data type names will always follow the format *somename\_Type*. The *structure\_list* consists of a number of data type definitions. An example of a complex data type could be:

\[
\text{Datatype Project\_Type}
\begin{array}{c}
\text{ProjectName\ of\ type\ String} \\
\text{ProjectDuration\ of\ type\ Integer}
\end{array}
\]

Finally, the *method\_list* is a list of methods or services that the object contains. The methods are what the object can do. In other words, the methods define the actions that the object may perform. Such actions usually make use of the object's attributes. A method may have parameters. The parameters are simply more information given to the method before it is called by the object.
5.5.8 Using the object template

To illustrate the above definitions with some practical examples, some objects will now be constructed. The simplest object is an object that has no function. This is the object that forms the basis for any other object that is created within the proposed model. Thus, using the basic template, an empty object could be constructed as follows:

Object Empty_Object

The object Empty_Object has no functionality, nor the ability to contain any information. There may be very few instances in which such an object would be useful - except for the derivation of new objects.

A more practical example is the MethodStep1_Object object, with the following definition:

Object MethodStep1_Object
   ProjectName_Attribute of type String
   ProjectDuration_Attribute of type Integer
   GetProjectName_Method
   GetProjectDuration_Method

A very simplistic representation of the initial phase of some methodology is given by the object MethodStep1_Object. This phase has only two stages, namely get the planned project's name and its estimated duration. To find these, the methods within the object are called. The functionality of each method could be to insert the values into the correct slot within the object's data area. The data areas are ProjectName_Attribute and ProjectDuration_Attribute respectively.

In order for this object to communicate its contents to other components of the proposed model, a message passing mechanism must exist. Such a mechanism is inherent in object-oriented system by way of the methods or services. Messages to an object can be likened to the traditional procedural paradigm where an explicit call was made to a procedure or function. A message to an
object takes the same approach: it is an instruction for some operation to be performed. The important difference is that the message or call is directed at an object. Thus, the object takes the responsibility for carrying out the action. Furthermore, the functionality exists within the object and is not a separate component.

The CASE tool component in the environment could contain the following object:

```
Object CASE_ERD_Object
  ProjectName_Attribute of type String
  EntityName1_Attribute of type String
  EntityName2_Attribute of type String
  RelationshipType_Attribute of type String
  GetRelationshipType_Method
```

This object defines a very simple entity-relationship diagram. The object contains three entities, namely `EntityName1_Attribute`, `EntityName2_Attribute` and `RelationshipType_Attribute`. The entity names are stored in the `EntityName` attributes of the object. The relationship between these two entities is stored in the `RelationshipType` attribute. To determine the relationship between these two entities, the method `GetRelationshipType_Method` is called.

The feedback loop's aim is to provide integration between two such objects as given above. It must be possible for the `MethodStep1_Object` and the `CASE_ERD_Object` to communicate and exchange information. This is the aim and purpose of the feedback loop.

### 5.5.9 Inter-object communication

[COAD90] explains the workings of a message to an object as follows: a message says which object is to perform the action. The message identifies what is to be done. Finally, the message gives arguments to be used for the execution of the selected method. The operation is named in the message. [SAVI90] defines these three parts of the message exchange among objects as the receiver of the message, the selector and the arguments. Both the receiver
and the selector must be specified, whereas the arguments are optional. A message is evaluated and the message is sent to the receiver. What is actually done is encapsulated in the object that receives the message. [COAD90] An object can change its state only upon receiving a message. [SAVI90]

How then would the object MethodStep2_Object receive information about the proposed duration of some software project? Clearly it cannot simply access the data storage areas, or attributes, of the object MethodStep1_Object, since this would be in contravention of the encapsulation property of that object. The message passing mechanism is formally defined as follows:

\[
\text{SenderObject} \text{ ReceiverObject}. \text{Method} \{ \text{Parameter} \_\text{List} \}
\]

The SenderObject sends a message to the ReceiverObject. The message is to execute a method named Method contained within the ReceiverObject. If necessary, parameters may be sent and received as well; the Parameter_List is optional.

Thus, to find the project's proposed duration, the following message passing scheme could be implemented:

\[
\text{MethodStep2\_Object} \text{ MethodStep1\_Object}. \text{GetProjectDuration\_Method}
\]

To store the data within MethodStep2_Object, a parameter such as MethodStep2_Object.Duration_Attribute could be added to the above message. In effect, this would store the result within the attribute Duration_Attribute of MethodStep2_Object. It should be clear that the called method does all the work; the SenderObject is not concerned how the method works, simply what the result is.

As the proposed feedback loop is concerned with the communication from a CASE tool component to a methodology component, a short example of such an inter-object communication will now be presented. It is important to understand that the object's location or base class should not interfere with the type of objects it may communicate with. Thus, if an object in the CASE component communicates with another object in the CASE component, this should be
possible. In the same way, an object in the CASE component should be able to communicate with an object in the methodology component. This is dependent only on the communications protocol used by the objects.

The example is similar to the previous one where the duration of some project was required. In this example, the requesting object is an object in the CASE component, such as the CASE_ERD_Object, and the source object is an object in the methodology component, such as the MethodStep1_Object. Once again, a method is called to retrieve the information; in this example the information required is the project name. It should be clear that encapsulation has an even more important task in this example, since information in a different and separated environment is being accessed. To retrieve the information, the following method call is used:

\[
\text{CASE_ERD_Object MethodStep1_Object.GetProjectName_Method}\ \\
\text{CASE_ERD_Object.ProjectName}
\]

Note that the parameter \( \text{CASE_ERD_Object.ProjectName} \) is added as a parameter to the method call. This defines the destination of the information sought. No other differences to previous examples should be present, except for the additional parameter and the fact that this is CASE to methodology communication. Using the objects defined here in a shared environment is relatively simple; the method calls are compatible.

Classes have been briefly mentioned before. To recap, classes are mechanisms that are found in all object-oriented programming languages. A class is used to define data structures and processing. In technical terms, a class is a static description of data and actions that may be performed on that data. An object is a run-time instance of a class.

The distinction between these two concepts can be likened to traditional variable declarations. Here, a data type is equivalent to a class. An instance or variable of a particular data type is a run-time instance of that data type. Once again, the technical implementation is not important, rather the theoretical aspects of the proposed model deserve attention.
Classes have more to them than simply being static descriptions of objects, however. Classes imply classification schemes. Thus, a number of objects may be derived from higher level objects. This is the concept of inheritance - classification and specialisation make this technique possible.

As an example, a diagram class in the CASE tool component may be defined as a high level object. The proposed model requires a number of different diagram types; each diagram type is different from any other in certain areas, but all share certain properties. Properties that are shared are defined in the top level class or object. To define a new diagram type, the high level diagram is taken as a template. By adding properties specific to the new diagram type an object can be easily created using the concept of inheritance. Inheritance does not apply solely to properties or attributes. In exactly the same manner as common attributes may be defined in a high level object definition, method definitions may also be defined. Objects derived from the high level object inherit all methods and may add new ones. As an aside, object-oriented programming languages allow overriding of inherited methods also. In this case, the definition of an inherited method is altered or extended to perform a different action.

5.5.10 Introducing inheritance

To illustrate the concept of inheritance, we shall return to the MethodStep example. Assuming that the MethodStep methodology is made up of five phases, it would be sensible to examine each of the phases to determine what they all have in common. Why one should look at commonality may not be clear immediately. According to [LOCH89], inheritance has many forms, depending on what we wish to inherit and when and how inheritance takes place. In most cases however, inheritance is strictly a reusability mechanism for sharing behaviour between objects. [LOCH89] The key word, reusability, should indicate the necessity for looking at commonalities among the objects.
To start, we define a base object. It is from this base object that all the others objects can be built.

\[
\text{Object MethodStepBase \_ Object} \\
\text{ProjectName \_ Attribute of type String} \\
\text{ProjectDuration \_ Attribute of type Integer}
\]

To keep the example simple, it will be assumed that we wish to know the proposed duration and the name of the project we are currently busy with in any phase of the methodology. It is not impossible that every object within both the methodology component as well as the CASE tool component require the same information. In this case, the message would have to be propagated throughout the environment by the feedback loop.

The first phase of the MethodStep methodology can thus be constructed from the base object. The base object forms the root or parent of an object class. In order to create the class, some new notation must be introduced to formalise the process of inheritance.

To indicate the inheritance of all attributes and methods from the base object \text{BaseObject} to some new object called \text{NewObject}, the following syntax is introduced:

\[
\text{Object NewObject (BaseObject)} \\
\{<\text{attribute\_list}>\} \\
\{<\text{method\_list}>\}
\]

Extending the notion of single inheritance, where some object inherits all properties of one object, to multiple inheritance, where some object inherits all properties from many objects, is easy:

\[
\text{Object NewObject {<(object\_list)>>} } \\
\{<\text{attribute\_list}>\} \\
\{<\text{method\_list}>\}
\]
An example of inheritance will follow shortly. In the definition, `object_list` may contain one or more object names. The inclusion of the `object_list` is optional. Thus, the above definition replaces the initial template.

To illustrate single inheritance by way of our example, the new object `MethodStep1_Object` inherits both attributes from the object `MethodStepBase_Object`.

```plaintext
Object MethodStep1_Object (MethodStepBase_Object)
  GetProjectName_Method
  GetProjectDuration_Method
```

It is important to note that the new object adds new features, in this case the two methods that were mentioned previously.

A second example displays the power of multiple inheritance in the CASE tool component. Here, a compound diagram object is created from two objects.

```plaintext
Object CASE_ERD_Object
  ProjectName_Attribute of type String
  EntityName1_Attribute of type String
  EntityName2_Attribute of type String
  RelationshipType_Attribute of type String
  GetRelationshipType_Method
```

This object has been introduced earlier. The second object is the `CASE_DFD_Object`:

```plaintext
Object CASE_DFD_Object
  ProjectName_Attribute of type String
  EntityName1_Attribute of type String
  EntityName2_Attribute of type String
  SourceEntityName_Attribute of type String
  GetFlowDirection_Method
```
CHAPTER 5: DERIVING A BASE FOR THE MODEL

The CASE_DFD_Object represents a very simple data flow diagram object, with two entities and the name of the entity from which the data emanates. This allows the method GetFlowDirection to report the direction of the data flow.

A new object can be formed by way of multiple inheritance, as follows:

```
Object CASE_Diagram_Object(CASE_ERD_Object, CASE_DFD_Object)
  ProjectName_Attribute of type String
  EntityName1_Attribute of type String
  EntityName2_Attribute of type String
  RelationshipType_Attribute of type String
  GetRelationshipType_Method
  SourceEntityName_Attribute of type String
  GetFlowDirection_Method
```

The new diagram object contains the detail from the CASE_ERD_Object as well as adding to that detail to create the CASE_DFD_Object.

5.5.11 Concluding the object-oriented approach

Two main points should be clear from the preceding discussion.

In the first place, attributes and services on the attributes are treated as an intrinsic whole - thus encapsulation results. This was illustrated by way of the examples in this chapter. The importance of encapsulation is that message passing is the only way to access some object. Every object maintains its own local state. [LOCH89] The result is that a fixed and well-defined interface exists for every object. This is a big advantage for the proposed model, where random access to any part of the model is not desirable.

However, encapsulation does have two disadvantages. In concurrent systems, certain data areas must be shared by more than one object. Interference results, because no one object controls the access to the data area. [LOCH89] This is not a problem for the proposed model, since it does not support concurrency. The second disadvantage is the problem of synchronisation.
Here, objects retain references to certain values. Since message passing is a dynamic operation, there is no guarantee when a certain object will pass a certain message. If the reference to some value still exists even though the value no longer exists, then inconsistencies may occur. [LOCH89] This may be a problem for the proposed model since a number of diagrams may reference the same information. If the information should change without the other objects being made aware of that fact, some inconsistencies may result. An effective solution is to place all information into a repository. Thus, the information exists only once. Every object must reference the repository to perform an action.

Secondly, the interface among objects is defined by the methods contained in the object. The preceding discussions centring on the encapsulation have already covered this point.

5.6 Integration of diagrams

The object-oriented approach is seen to be the main focus and building block of the model. Diagrams are seen as an important part of the overall design process and are almost always present in both CASE tools and methodologies. Thus, the proposed feedback loop must also be able to cope with the information exchange to and from diagrams.

The information exchange from the diagrams in the CASE tool component to the methodology component is especially important in cases where the methodology requires certain diagrams to be created. Once a diagram is created, its creation must be registered with the methodology, since other diagrams may need to interact with the diagram, or require relationships to be set up to that diagram. Without information exchange between the two components, this registration cannot happen and can hamper the design process.

Diagram integration through modelling transparency is introduced as a supporting approach to the model's development. It is important to note that this approach relies fully on the object-oriented approach as a base and makes use of the functionality provided by that approach. Thus, the diagram integration approach is used to further motivate the use of objects.
5.6.1 Integration of diagrams through modelling transparency

Modelling transparency is defined by [BRIN93] as a term that reflects the need for coherent support for modelling tasks of requirements engineering. In more precise terms, modelling transparency refers to the possibility of changing attributes and manipulating relationships within diagrams. Thus, the diagrams and diagramming techniques are transparent to each other.

The concept of diagram integration through modelling transparency as presented by [BRIN93] is an important building-block of the proposed model. In the paper, [BRIN93] introduces modelling transparency as the functionality of CASE tools that supports relating diagrams to each other. [BRIN93] introduces the concept of various diagram editors that interact with each other. Interaction takes place via some repository.

In current CASE tools, the lack of integration of various tools, or parts thereof, require well co-ordinated use and accurate clerical work. [BRIN93] The integration of CASE tools is very important to reconcile results gained from each tool. As diagrams play a very important part in the design of any system, [BRIN93] views a CASE tool as a set of diagram editors that store and retrieve diagrams and information in a common repository.

Viewing a CASE tool as a set of diagram editors is dependant on the type of CASE tool one is examining. Elsewhere in this study, various tool sets and tool groups were discussed. Rapid prototyping tools, code generation tools and diagram related tools were touched upon. Since these were only tools, or tool subsets, it would not make sense to call them CASE tools. Given the scope of ones requirements, however, such a tool could fulfil the same requirements as could a fully fledged integrated CASE (ICASE) tool. Clearly then, one must narrow or broaden one's scope depending on the environment in question.

The model developed in this chapter attempts to provide an ideal environment for the feedback loop. In other words, it provides the base to which both the CASE and the methodology component should ideally conform. The major component of this study is the feedback loop. This is a mechanism for combining a CASE tool and some methodology. As was already mentioned,
which CASE tool is being implemented on one side, and which methodology is implemented on the other side is irrelevant. The aim of the study is simply to illustrate the possibility of interfacing the two components in a sensible fashion. Even though it has already been touched upon here, the feedback loop is the subject of the next chapter.

5.6.2 The role of diagrams

The view proposed by [BRIN93] directly supports the view taken in the development of the proposed model. Diagrams are very important for both analysis and design of systems. Also, diagrams are easy to understand and represent some real-world situation. Using simple diagramming techniques, supplemented where necessary by notes and discussions with the developer, makes it possible for the user to be involved at every stage of the construction, providing confirmation, comment and feedback on technical decisions. [MAIR93] Furthermore, diagrams allow specialists and non-specialists from different fields to communicate more effectively.

Most tools that provide an abstraction of the designed system include a formal description language which imposes an inevitable syntactic overhead on the designer. A graphical user interface and use of diagrams contributes to the enhancement of the designer's productivity and efficiency, as well as helping to eliminate errors. [COHE92] Whether or not the view taken by [BRIN93] defines a true CASE environment is a question that is beyond the scope of this study.

A CASE tool should support at least some diagramming technique which is usually derived or driven by a chosen methodology. Thus, there is an interdependency between the CASE tool and the methodology. It is usual that a subset of the following diagrams or diagramming techniques is supported:

- entity-relationship diagrams (ERD)
- data flow diagrams (DFD)
- structure charts (SC)
- control flow diagrams (CFD)
The interrelationship among the various diagrams is an important part in the modelling of the system being designed. The reason for this is simple. A data flow diagram represents different facts concerning the system under design than a structure chart, for example. Even though the system being represented is the same one, different diagram types represent different views and functions of the system. In order to ensure that the system is designed correctly, all aspects must be considered.

Diagramming is a well-known structured method. Virtually all structured methods rely on diagrammatic techniques to achieve their objectives, replacing pages of technical jargon with easy to understand diagrams. [MAIR93] The interrelationship among diagrams is important as every diagram models only a certain view of the problem area. Whereas a data flow diagram models flows of data within the problem area, an entity-relationship diagram models the entities within the problem areas and how they interact. Both diagrams are essential for the successful completion of the development task. Yet to be absolutely useful, these diagrams must be merged to create a full view of the entire problem space. Creation of diagrams that map only certain areas of the problem space are helpful only for purposes of abstraction.

The process of changing, adding and removing elements and relationships from any diagram should be supported by the proposed model. The interrelationship among diagrams must be defined further. In fact, two separate issues must be recognised:

1. interrelationship of different diagram types within the CASE tool component. In this instance, the CASE tool component is singularly responsible for propagating changes from one diagram type to another. In most cases, this occurs by way of a repository. This type of interrelationship is not the function of the feedback loop

2. exchange of information from a diagram in the CASE tool component to an object in the methodology component of the environment. In this instance, the changes made in the diagram on the CASE tool component must be conveyed to the suitable phase in the methodology component. Similarly, a change in the methodology component's object must be conveyed to the appropriate diagram in the CASE tool component. This is the function of the feedback loop.
A short definition of modelling transparency was given at the beginning of this section, but can now be expanded. Because of the resulting transparency of the diagrams and the diagramming techniques in relation to each other, this functionality of CASE tools is named *modelling transparency*. [BRIN93] Modelling transparency does not imply that no relationships exist among diagrams however, as the following illustration shows.

![Figure 5.1 - Modelling transparency and diagram relationships](image)

It will be shown that diagrams in their native form are not necessarily suitable base elements for the proposed feedback loop. Instead, and as has been described, objects will be used to create the base for the proposed model. Diagrams can then be created and built from the existing objects.
5.6.3 Modelling transparency

The term *modelling transparency* was selected to reflect the need for coherent support of the modelling tasks of requirements engineering. Transparency indicates the absence of intermediate user operations to transfer changes from one diagram to another. The *degree of transparency* refers to the extent to which intermediate operations may be needed. [BRIN93] The more intermediate operations must take place, the lower the transparency is said to be.

The high degree of relationships found in many systems creates an intricate web of dependencies among the diagrams of the requirements specification. Relationships must exist among entity types, processes and modules due to the nature of the systems being modelled. Systems or problem spaces being modelled are usually complex.

Airline reservation systems or shop floor automation software are two examples of very complex software implementation projects. Relationships among various components of such systems act as the glue that holds everything together. Without relationships, there would be no need to interact with other parts of the system or organisation. Once interaction is not required, the feasibility of the proposed software implementation project falls into question.

Dependencies caused by relationships must, therefore, be modelled as well. Once displayed as a set of diagrams, the relationships take on various meanings. As an example, the relationships in a data flow diagram have a different meaning to those in an entity-relationship diagram. To model these relationships, the mechanism of message passing among various objects will be employed. This coupled with a global repository that contains all relevant information should provide a solid foundation for the representation of relationships.
5.6.4 Dependencies among diagrams

According to [BRIN93], two types of dependency exist. These are sequential dependencies and parallel dependencies. Both are important as they influence the relationship and interaction among two or more diagrams. Relationships are intrinsic to modelling diagrams. Thus, the feedback loop must provide a method to reconcile changes made in any part of the design process. The methodology component must be made aware that links among objects have been made, in exactly the same way as the CASE component must be made aware of newly created, or existing links. In so doing, the methodology component can refresh the status of certain tasks and initiate the next task or phase.

A diagram illustrates how such an exchange of information could be accomplished by making use of a repository. All diagrams, or objects, have access to one single repository. Information is shared and distributed by objects in this repository.

![Diagram](image)

Figure 5.2 - Information sharing using a repository

The terms sequential and parallel dependency will be explained briefly, together with their usefulness within the proposed model.
5.6.5 Sequential dependency

A diagram is said to be sequentially dependent on another, already defined diagram if its components need to be brought into agreement with components in the other diagram. [BRIN93]

Assume that some ERD has already been established, containing a number of entities. Corresponding entities must be created to describe the entities within that diagram, for example the Customer and Article entities. For each of these entities, attributes will be provided. Should a departure from the sequential dependencies be necessary, the already established diagram would have to be changed first.

In view of the proposed model, the statement made by [BRIN93] should be expanded. For purposes of this study, an object is said to be sequentially dependent on another already defined object if its components need to be brought into agreement with components in the other object. Note that the emphasis no longer rests solely on the diagram, but rather on the concept of objects.

As an example, the methodology component may require the creation of a data flow diagram. The feedback loop is instructed to inform the CASE tool component of this fact. The user creates the DFD. At this point in time there is an inconsistency within the environment. Using the terminology introduced in this section, one or more objects in the methodology component are sequentially dependant on one or more objects in the CASE tool component. The reason for this is that the CASE tool component contains information which must be propagated to the methodology component in order for the environment to be synchronised again. Thus, the CASE tool component instructs the feedback loop to pass the new information on to the methodology component. Once the information has been absorbed and distributed in the methodology component, the environment is in a consistent state until the next change is made in one of the two components.

In the same way, the environment could be inconsistent if a change is made within the methodology component. As an example, a developer may decide to
skip a step in the methodology due to a deadline that cannot be met. Since the CASE tool component is not yet aware of this fact, it may expect the construction of an ERD due to a previous instruction from the feedback loop. If the developer continues with the next step in the methodology, such as the creation of an SQL script for a database table, the environment is inconsistent. Once again there are one or more objects that are dependant on each other. As soon as the feedback loop instructs the CASE component of the change, the environment is once again consistent.

How does this new definition of sequential dependency develop the aim of the proposed model? As has been stated before, objects represent real-world entities. A diagram is nothing more than a representation of a real-world entity. Thus, a diagram can be likened to an object. Taking this a step further, a diagram could be represented by an object at a lower level. This lower level is not the level at which the user works, but is an internal representation of the diagram. In this way, the diagram contains information and possibly even methods that may be used to carry out some action. Such methods could be methods to propagate information about these objects to the feedback loop. The feedback loop could then pass this information on to objects in the methodology component. The distributing of information within such an environment could be accomplished by making use of a global repository, as was illustrated earlier.

Diagrams are used as the main example throughout this section since this section deals with diagrams and modelling transparency and because the base of [BRIN93]'s theory revolves around diagrams. It is important to realise that the diagrams can be seen as higher level representations of objects.

As an example, reconsider the MethodStep example introduced earlier. Once one instance of the object MethodStep1_Object has been derived from the base or parent object, it is immediately usable. Any number of other objects, such as MethodStep2_Object, may be derived from the base object. Once the new objects have been created, a sequential dependency exists between the new object and the base object. This is simply a result of the inheritance mechanism discussed earlier. If the base object changes, all objects derived from it are no longer consistent with the original structure and classification scheme. Thus, the derived objects must be modified in accordance with the changes made to the
base object. If compared to the previous example dealing with sequential dependencies for diagrams, it should be clear that the new definition is more powerful and allows any type of structure to be sequentially dependant on another related structure.

As an aside, the object that stores all information for a diagram is necessarily very complex. The various relationships, entities and attributes, together with other information that must be stored requires a very flexible structure. It would be useful to create a classification scheme and associate this scheme with the diagram object. By extracting all commonalities, a base object should be constructed. To create an object for a certain diagram type, multiple inheritance techniques should be used to draw together all relevant object and data types.

5.6.6 Parallel dependency

Two diagrams are dependent in parallel if they model properties of a particular set of artefacts, and are developed in parallel with each other and are more or less simultaneously established. [BRIN93] Such a dependency may exist between two diagrams of the same or different type.

For example, some entity types may be shared by the entity relationship diagrams that model the data of two distinct processes in one application. The definitions of these entity types, as well as their attributes and data types should be in agreement with each other or be brought into agreement. [BRIN93]

Once again, the definition given by [BRIN93] should be expanded to include the object-based approach used for the proposed model. Thus, two objects are dependent in parallel if they model properties of a particular set of artefacts, and are developed in parallel with each other and are more or less simultaneously established. [BRIN93]

When examining the MethodStep example, two or more objects may be derived from the base object. Assume three objects are created, namely MethodStep1_Object, MethodStep2_Object and MethodStep3_Object. Each object instance has added some attributes and methods specifically for its own
use. Thus, each of the three objects is unique in some way. Commonality among the objects is achieved since they are all derived from the base object; they therefore contain attributes and methods common to the base object. A parallel dependency exists among all three objects due to this commonality. If information for a common attribute changes, this change must be reflected in all other objects sharing this attribute.

[BRIN93] states that such a dependency may exist between two diagrams of the same or different type. For purposes of the proposed model, it must be noted that even though objects may be structurally and operationally different, a parallel dependency can exist among objects only if some attribute or method is common to all the objects participating in the dependency. This means that objects in a class could at some stage be involved in a parallel dependency. Since objects and classes in the CASE tool component and the methodology component are different, parallel dependencies cannot occur among objects of the environment. The feedback loop must be capable of dealing with sequential dependencies only.

5.6.7 Degrees of modelling transparency

[BRIN93] distinguishes four degrees of modelling transparency. Degrees of modelling transparency are usually defined in terms of functionality of a particular tool. This tool is usually an automated or computerised CASE tool.

Stand-alone CASE tools with a non-accessible repository do not support modelling transparency. Users of such tools are required to keep track of diagram and structure changes themselves, and apply these to all relevant parts. This is modelling transparency degree 0.

Modelling transparency degree 1 is assigned to CASE tools with an open, accessible repository which enables dependencies to be established. Such links can be shown only by exiting from the current diagram editor and starting up another. Due to this time consuming process, the user will still handle many changes to various diagrams manually.
Some CASE tools offer switching among a number of diagram editors. Usually, it is possible to display two diagrams side by side, especially in the now standard graphical environments. Such CASE tools support modelling transparency degree 2.

Modelling transparency degree 3 is offered by those CASE tools providing hypertext functionality. Such tools are called Hyper-CASE tools. In a Hyper-CASE tool, the user may define links from each component in a certain diagram to any other component in another diagram at will. The related diagrams become accessible to each other and dependencies may be traversed in any arbitrary way.

5.6.8 Degrees of modelling transparency and the feedback loop

Modelling transparency is also an issue for the proposed feedback loop. A global repository is part of the feedback loop - thus, modelling transparency of degree 0 is not applicable. In view of the proposed model, modelling transparency of degrees 1 and 2 are also not an issue. These deal only with diagrams and, as has been explained, how and when diagrams are displayed is a question of implementation. For all intents and purposes, the repository structure together with the single access layer and the object message passing mechanism provides modelling transparency of degree 3. It must be noted that objects are the base of the model, not diagrams. Diagrams are simply a means of representing an object, as has been mentioned previously.

The only degree of modelling transparency that is of use to the feedback loop is modelling transparency of degree 3. The reason for this is that the feedback loop must have access to all information at once. Furthermore, the feedback loop mechanism must have a global view of the data. Thus, any one of the two components of the model environment are allowed access to any portion of the information stored within the other component. This access is handled by way of the feedback loop. To prevent any inconsistencies, the feedback loop requires a mechanism by which it can determine which information is new and which is not.
As has been mentioned, the feedback loop should have access to an openly-accessible repository for information sharing. The feedback loop is the owner and administrator of this repository. All changes made to the repository are reflected on-line; this means that dependencies built up within the repository are immediately updated. Since the proposed model deals with objects, the status of each object is refreshed within the repository. Whether or not some diagram editor is updated on-line or only after exiting and re-entering it, is a matter of implementation. Such functionality can be added rather easily, however, once the underlying repository behaves in the above fashion.

The Hyper-CASE extension refers to so-called "hot links" among the various elements of the repository. Since all objects that are stored in the repository form part of some classification hierarchy, hot links do exist. Any change in a base object must result in a change in the derived object. This is a feature of inheritance, as well as sequential dependencies.

5.6.9 The proposed model and the integration of diagrams

The use of diagrams for the proposed model must be explained. Even though it has been said that diagrams are not a base for the proposed model, the importance of diagrams must not be ignored. Through the use of diagrams, a great deal of information can be stored and manipulated. Furthermore, diagrams are a better means of representing modelling data, especially when this data must be communicated to a person.

Diagrams are the best way of representing the complex relationships that exist in system to be modelled. A diagram is more readily understood than a list or a long explanation. Object templates and object definitions are a representation that are useful only for operations internal to the model. Externally, the model must translate object definitions, and especially object contents, into a form easily understood by a software engineer or developer. Diagrams are a natural choice.

A number of different diagrams exist for modelling and representing systems. The best known are entity relationship diagrams (ERD's) and data flow diagrams
(DFD's). As has been discussed earlier, the underlying structure for each diagram and diagram type is an object hierarchy or classification scheme. Such a classification has a number of base objects. To create a specific object, for example the object \textit{ERD\_Object}, all relevant base objects would be employed to create this object:

\begin{center}
\textit{Object ERD\_Object (Base1\_Object, Base2\_Object, Base3\_Object, ...)}
\end{center}

The creation of the base objects must be a carefully considered operation. Only those attributes and methods that are common to all types of diagrams must be included.

Each diagram is specific to a subset of the solution to a problem; the problem being the system being modelled. Thus, each diagram contains different information; all diagrams together can describe the system completely. Only a number of different diagram types need to be used. The diagrams reside within the repository of the CASE tool in the environment. The number and type of diagrams in use are determined by the CASE tool, the methodology and the user. Thus, the feedback loop must have the ability to convey information to and from any type of diagram. This can be accomplished quite simply by using the object-oriented representation of the diagram. As was noted earlier, the object-oriented description or representation of a diagram is simply a different view of the same diagram.

The dependencies involved among the diagrams occur naturally. This is due to the varying nature of the diagrams. Whereas a DFD focuses on data flows, the ERD focuses on entities and their relationships. Both diagrams pertain to the same objects, however. The maintaining of these dependencies can be handled most easily by a high degree of modelling transparency within the CASE tool component. Thus, a change to one diagram should effect a change in all others.

The feedback loop makes use of a data repository to store data pertaining to its function. Thus, the interchange of diagram data occurs via this repository. Objects within the repository are updated automatically via the feedback loop's functionality. By interrogating the repository, the feedback loop can retrieve
information specific to any object or any data structure. If links exist among elements within the repository, these are conveyed by message passing mechanisms. The feedback loop keeps track of all updated objects, as well as the source and destination components. The source and destination components are the CASE tool component and the methodology component.

5.7 Conclusion

This chapter has served as a vehicle to define an ideal environment in which the feedback loop can operate. It was shown that the object-oriented approach is helpful, if not essential, to create this environment. Furthermore, it lends objects with inheritance and message passing mechanisms to the model. Communication by message passing was shown to be the basis by which objects can cause actions. Inheritance allows the creation of new objects by making use of existing objects.

The integration of diagrams was also discussed. Instead of relying on the diagramming approach as a means of creating the environment for the feedback loop, the diagramming approach was based fully on the object-oriented approach. Thus, the feedback loop does not have to contend with various types of information, but deals only with objects - be they diagrams or anything else. Diagramming lends the sharing of relationships and dependencies among different views of the same problem space to the proposed model.

The next two chapters make use of the ideas developed in this chapter to discuss the CASE and methodology components in detail. Given the definitions made here and in those chapters, the feedback loop's functions and its operation can be defined more easily and precisely.
6.1 Introduction

The previous chapter has defined the building blocks that will be used to create the model in this study. In that chapter, the object-oriented paradigm was explained and chosen as the technology upon which the model is based. [MULL90] supports this by stating that more attention has been paid to object-orientation as a technology in its own right.

In this chapter, the CASE tool component will be defined in object-oriented terms. The CASE environment developed and explained here is a small subset of a real CASE environment. The subset chosen should be sufficient to describe the feedback loop in detail later on.

6.2 Defining the CASE tool component

The CASE tool component is the component which interacts directly with the user, whereas the methodology component has a more passive role. The functions provided by the fictitious CASE tool component in this chapter are the following:

1. creation and modification of diagrams for system modelling
2. creation and modification of repository structures for the system being modelled, such as database tables and SQL commands

These functions will now be discussed and defined.
6.2.1 Creation and modification of diagrams

The use of diagrams is very important in the design and modelling of a proposed software system. Thus, the information model is produced in a graphical form that provides a global view of the conceptual entities in the problem space. [MAND92] The following diagrams will be supported by the model CASE tool in this chapter:

a) entity-relationship diagrams (ERD)

b) data flow diagrams (DFD)

a) The entity-relationship diagram

The entity-relationship model was proposed by Peter Chen in 1976. It has been accepted as a graphical approach to modelling relationships among entities. [PRAT87] The use of the entity-relationship model has extended to become a database design tool. In the entity-relationship diagrams, entities are represented as rectangles and relationships as diamonds. Lines connect the entities and relationship symbols. The lines are labelled, depending on the type of relationship they represent. Relationships can be one-to-one, one-to-many or many-to-many.

The following figures shows an entity-relationship diagram with a one-to-many relationship. Note that the relationships may be abbreviated to 1-1 (one-to-one), 1-N (one-to-many) and M-N (many-to-many).

![Entity-Relationship Diagram](image)

Figure 6.1 - Relationships in an entity-relationship diagram
In order to ensure that this diagram is useful for database design and creation, attributes are added to the entities. Thus, an entity-attribute-relationship diagram results. Attributes are attached to entities and are represented by circles. The attributes define the information attached to the entity. The following diagram shows an example of an entity-attribute-relationship diagram.

It will be seen that the inclusion of attributes ties directly into the object-oriented approach chosen for the creation of the environment.

**b) The data flow diagram**

A data flow diagram is a graphical notation which describes the functions to be carried out by the system being modelled. [INCE89] Data flow diagrams consider the flow of information within the system being modelled or described. In other words, the data flow diagram identifies specific functions or processes, data stores and the flow of data between processes and data stores. [MAIR93] It is typical to decompose processes identified in a high level DFD into lower DFD’s to show more detail. The purpose of each DFD remains the same though, to display the context of the system to the outside world. [MAIR93]
CHAPTER 6: DEFINING THE CASE ENVIRONMENT

The following illustration shows a data flow diagram with the same entities introduced earlier for the ERD example.

![Data Flow Diagram]

Figure 6.3 - Data flow diagram

The purpose of both types of diagrams to be included in the CASE tool component have been discussed. Derivation of the objects to represent these structures is the next step.

6.2.2 Deriving the diagram object class

The illustrations provided during the discussion of both the entity-relationship and data flow diagrams provide some insight into the requirements necessary for the diagram object class. As is customary, an empty object with no methods and attributes is defined. This object forms the parent of the diagram object class and is named **CASE_Diagram_Parent**, since every object in the diagram class will be derived from it or one of its children. The structure of the object is as follows:

```
Object CASE_Diagram_Parent
```
This object is the uppermost in the diagram class hierarchy.

In order to provide information as to the diagram being created, a new object is derived. This object provides mainly administrative information, such as the creation date of the diagram, the version of the diagram and the name of the person creating or modifying the diagram.

In order to interface to this information, two methods are provided for each attribute. The naming convention for one of these was introduced in the previous chapter. This was the method with theprefix Get. All methods having this prefix allow another object to retrieve information concerning the attribute named in the Get method name. For example, the method GetDiagramVersion_Method allows another object to retrieve the contents of the attribute DiagramVersion_Attribute. To change the contents of each attribute, methods with the prefix Set are provided. Thus, the method SetDiagramVersion_Method would allow another object to modify the contents of the attribute DiagramVersion_Attribute.

It may be felt that the attributes provided in this object are not sufficient. As an example, it may be felt that the details that can be associated with a diagram should include the project name and a number for the task associated with a project plan detailing the software development project. For purposes of this study however, the details selected are assumed to be sufficient to illustrate the operation of the proposed model.
The `CASE_DiagramDetail_Object` definition is as follows:

```
Object CASE_DiagramDetail_Object(CASE_Diagram_Parent)
  DiagramVersion_Attribute of type String
  DiagramDateModified_Attribute of type String
  DiagramCreatedBy_Attribute of type String
  SetDiagramVersion_Method
  GetDiagramVersion_Method
  SetDiagramDateModified_Method
  GetDiagramDateModified_Method
  SetDiagramCreatedBy_Method
  GetDiagramCreatedBy_Method
```

The most important thing for a diagram object to "know" is the type of structure it is representing. Thus, a new object is added which includes this information together with methods to manipulate it:

```
Object CASE_DiagramType_Object(CASE_DiagramDetail_Object)
  DiagramType_Attribute of type String
  SetDiagramType_Method
  GetDiagramType_Method
```

The `CASE_DiagramType_Object` is a parent for three subclasses in the diagram class hierarchy. The definition of this object is general enough to provide a base for the objects `CASE_Relationship_PARENT`, `CASE_Entity_Object` and `CASE_Flow_Parent`. 
6.2.3 Deriving a subclass for entities and attributes

The creation of a more generic and useful object continues once one has determined the most important pieces of information all derived objects should contain. The issue is to find the most common denominator and place this into a new base object. Since every diagram contains entities, it seems natural to add attributes to hold entity information together with methods to manipulate the entity information:

Object CASE_Entity_Object(CASE_DiagramType_Object)
   EntityIndex_Attribute of type Integer
   EntityName_Attribute of type String
   SetEntityIndex_Method
   GetEntityIndex_Method
   SetEntityName_Method
   GetEntityName_Method

The CASE_Entity_Object contains information concerning entities in a diagram. The diagram may be either a data flow diagram or an entity-relationship diagram. This is logical since both diagrams contain and deal with entities. However, entities are not the only components in these diagrams. As was illustrated during the discussion of the two diagram types, attributes are often required. It was shown that these attributes are attached to specific entities. Thus, an object is derived from the CASE_Entity_Object to allow the inclusion of attributes.
The new object is named `CASE_EntityAttribute_Object`.

```
Object CASE_EntityAttribute_Object(CASE_Entity_Object)
    EntityAttributeIndex_Attribute of type Integer
    EntityAttributeName_Attribute of type String
    EntityParentIndex_Attribute of type Integer
    SetEntityAttributeIndex_Method
    GetEntityAttributeIndex_Method
    SetEntityAttributeName_Method
    GetEntityAttributeName_Method
    SetEntityParentIndex_Method
    GetEntityParentIndex_Method
```

A short explanation of the attributes in both the `CASE_Entity_Object` and the `CASE_EntityAttribute_Object` is in order. If one examines any ERD it will be clear that there are one or more entities. The same is true for any DFD. To distinguish these entities by name is enough for the person creating the diagram. To represent the entities within the object however, is best accomplished by assigning a number, or index, to each new entity introduced. The reason for this will become clear shortly. Thus, the `CASE_Entity_Object` makes provision for both a name and an index for the entities. As usual, `Get` and `Set` methods are provided for both attributes.

In the same way as there are one or more entities for each diagram type, so each entity may have one or more attributes attached to it. The object `CASE_EntityAttribute_Object` provides a name and an index for every attribute, together with `Get` and `Set` methods. To link attributes to the appropriate entity, an extra field is provided in the `CASE_EntityAttribute_Object`. The attribute `EntityParentIndex_Attribute` contains the index to the entity it is attached to.
For example, an instance of the CASE_Entity_Object may contain the following values:

\[
\begin{align*}
\text{EntityIndex\_Attribute} & = 2 \\
\text{EntityName\_Attribute} & = \text{CUSTOMER}
\end{align*}
\]

The CUSTOMER entity has the attributes CODE and NAME. Two instances of the CASE_EntityAttribute_Object are required - one for each of the attributes. The first instance for the attribute CODE would contain the values:

\[
\begin{align*}
\text{EntityAttributeIndex\_Attribute} & = 1 \\
\text{EntityAttributeName\_Attribute} & = \text{CODE} \\
\text{EntityParentIndex\_Attribute} & = 2
\end{align*}
\]

The second instance for the attribute NAME would contain the values:

\[
\begin{align*}
\text{EntityAttributeIndex\_Attribute} & = 2 \\
\text{EntityAttributeName\_Attribute} & = \text{NAME} \\
\text{EntityParentIndex\_Attribute} & = 2
\end{align*}
\]

It is important to understand how the attributes CODE and NAME are linked to the entity they are describing. The EntityParentIndex\_Attribute with index value 2 links to the EntityIndex\_Attribute with value 2. The attributes within the entity may be numbered in any way, although sequential numbering is preferable.

The common components of both the data flow diagrams and the entity-relationship diagrams have been defined. The actual ERD and DFD objects can now be defined.
### 6.2.4 Deriving a subclass for entity relationship diagrams

The parent object for the ERD subclass is named `CASE_Relationship_Parent`. It forms the base for the `CASE_ERD_Object` by inheriting all necessary diagram related information from the `CASE_DiagramType_Object`. Thus, the `CASE_Relationship_Parent` contains information regarding the diagram name, diagram type, date created and diagram version to name a few.

Object `CASE_Relationship_Parent(CASE_DiagramType_Object)`

The components involved when creating an entity-relationship diagram are entities, attributes and relationships between various entities. The entity and attribute information can be found in the objects `CASE_Entity_Object` and `CASE_EntityAttribute_Object` respectively. To make provision for relationships in an ERD, the object `CASE_Relationship_Object` is created. The sole purpose of this object is to represent relationships between two entities. The definition is:

Object `CASE_Relationship_Object(CASE_Relationship_Parent)`

- `RelationshipIndex_Attribute` of type Integer
- `RelationshipName_Attribute` of type String
- `FromEntityIndex_Attribute` of type Integer
- `ToEntityIndex_Attribute` of type Integer
- `SetRelationshipIndex_Method`
- `GetRelationshipIndex_Method`
- `SetRelationshipName_Method`
- `GetRelationshipName_Method`
- `SetFromEntityIndex_Method`
- `GetFromEntityIndex_Method`
- `SetToEntityIndex_Method`
- `GetToEntityIndex_Method`
CHAPTER 6: DEFINING THE CASE ENVIRONMENT

This object makes use of the same linkage structure as introduced for the 
CASE_EntityAttribute_Object. In that object, a link from an attribute to an entity 
was created by including the index of the entity together with the attribute 
pertaining to that entity. The relationship has a name and an index together with 
Get and Set methods. Once again, the attribute RelationshipName_Attribute is 
provided to describe the relationship between two entities. The attribute 
RelationshipIndex_Attribute is used for internal numbering purposes.

To indicate the relationship between two entities, the attributes 
FromEntityIndex_Attribute and ToEntityIndex_Attribute are provided. The 
attribute FromEntityIndex_Attribute contains the index of the first entity and the 
attribute ToEntityIndex_Attribute contains the index of the other entity. Both 
attributes have appropriate Get and Set methods for maintaining their values.

To illustrate how relationships are created, assume that two entities exist, 
namely SALESREP and CUSTOMER. Thus, two instances of the 
CASE_Entity_Object exist. One for the SALESREP entity, the SALESREP 
instance:

\[ EntityIndex_Attribute = 1 \]
\[ EntityName_Attribute = SALESREP \]

and one for the CUSTOMER entity, the CUSTOMER instance:

\[ EntityIndex_Attribute = 2 \]
\[ EntityName_Attribute = CUSTOMER \]

The attribute EntityIndex_Attribute in both instances is important for the 
relationship to be created.
CHAPTER 6: DEFINING THE CASE ENVIRONMENT

To create the relationship between the SALESREP and CUSTOMER, an instance of \textit{CASE\_Relationship\_Object} is created. The instance would contain the following values:

\begin{itemize}
  \item \texttt{RelationshipIndex\_Attribute} = 1
  \item \texttt{RelationshipName\_Attribute} = SALESREP-CUSTOMER
  \item \texttt{FromEntityIndex\_Attribute} = 1
  \item \texttt{ToEntityIndex\_Attribute} = 2
\end{itemize}

The index given to the relationship allows the relationship to be maintained internally. The name is descriptive and may be used for documentation purposes. The attribute \texttt{FromEntityIndex\_Attribute} contains the same value as the attribute \texttt{EntityIndex\_Attribute} for the SALESREP instance, namely 1. In the same way, the attribute \texttt{ToEntityIndex\_Attribute} contains the same value as the attribute \texttt{EntityIndex\_Attribute} for the CUSTOMER instance, namely 2. It should be clear how the SALESREP-CUSTOMER relationship is brought about.

As an aside, it should also be clear that the values transferred to the appropriate object's attributes make use of the \texttt{Get} and \texttt{Set} methods provided in the source object.

Before deriving the definition for the \textit{CASE\_ERD\_Object}, one more object is required. This object "knows" what type of relationship has been created between two objects:

\begin{itemize}
  \item \texttt{Object CASE\_RelationshipType\_Object(CASE\_Relationship\_Object)}
    \item \texttt{RelationshipType\_Attribute} of type \texttt{String}
    \item \texttt{SetRelationshipType\_Method}
    \item \texttt{GetRelationshipType\_Method}
\end{itemize}
By making use of this object, the example above may be expanded to include the type of relationship created between the SALESREP and the CUSTOMER entities. Thus, the attribute RelationshipType_Attribute is given a value describing the relationship between the entities:

\[ \text{RelationshipType\_Attribute} = 1-N \]

Once again, Get and Set methods are provided for this attribute.

With all prerequisites for the ERD object fulfilled, the object CASE\_ERD\_Object may be created. Since this object requires information from the two subclasses CASE\_Relationship\_Parent and CASE\_EntityAttribute\_Object, multiple inheritance is employed to create the object CASE\_ERD\_Object:

\[ \text{Object CASE\_ERD\_Object(CASE\_EntityAttributeObject, CASE\_RelationshipType\_Object)} \]

6.2.5 Deriving a subclass for data flow diagrams

The derivation of the data flow diagram subclass follows the same path as the derivation for the entity relationship diagram subclass. The structure of the objects within this subclass is obviously different.

The parent object for the DFD subclass is named CASE\_Flow\_Parent. It forms the base for the CASE\_DFD\_Object by inheriting all necessary diagram related information from the CASE\_DiagramType\_Object.
CHAPTER 6: DEFINING THE CASE ENVIRONMENT

The CASE_Flow_Parent contains information regarding the diagram name, diagram type, date created and diagram version as did the CASE_Relationship_Parent object.

Object CASE_Flow_Parent(CASE_DiagramType_Object)

Whereas the CASE_ERD_Object deals with relationships between entities, the CASE_DFD_Object deals with the flow of information between entities. To enable the object to deal with data flows, an object CASE_FlowType_Object is created.

Object CASE_FlowType_Object(CASE_Flow_Parent)

FlowIndex_Attribute of type Integer
FlowType_Attribute of type String
FlowFromIndex_Attribute of type Integer
FlowToIndex_Attribute of type Integer
SetFlowIndex_Method
GetFlowIndex_Method
SetFlowType_Method
GetFlowType_Method
SetFlowFromIndex_Method
GetFlowFromIndex_Method
SetFlowToIndex_Method
GetFlowToIndex_Method

An explanation of this object is required even though the structure is very similar to that of the CASE_Relationship_Object. The attribute FlowIndex_Attribute assigns an index to uniquely distinguish this data flow from any other data flow. The attribute FlowType_Attribute provides storage for a descriptive name.
The attributes FlowFromIndex_Attribute and FlowToIndex_Attribute are used to link two entities involved in the data flow. The concept of the link is identical to the link used for the CASE_ERD_Object's entities and attributes.

As an example, consider the entity SALESREP introduced earlier. Assume that a table TBL_SALESREP exists in some database. The SALESREP information is to be stored in that table. To apply the concepts developed in this chapter, two entities exist, namely TBL_SALESREP and SALESREP. Two instances of the CASE_Entity_Object must be created. The first is the TBL_SALESREP instance with the following values:

\[
\text{EntityIndex}\_\text{Attribute} = 1 \\
\text{EntityName}\_\text{Attribute} = \text{TBL\_SALESREP}
\]

The SALESREP instance contains the values:

\[
\text{EntityIndex}\_\text{Attribute} = 2 \\
\text{EntityName}\_\text{Attribute} = \text{SALESREP}
\]

To create the definition for the data flow between these entities, one instance of the CASE_FlowType_Object is required. The attributes are instantiated with the correct values retrieved via the method GetEntityIndex_Method from both instances of the CASE_Entity_Object. The value retrieved is placed in either the FlowFromIndex_Attribute or the FlowToIndex_Attribute in the instance of the CASE_FlowType_Object, depending on the direction of the flow. In this example the flow is from the SALESREP instance to the TBL_SALESREP instance.
Thus, the instance of the \texttt{CASE\_FlowType\_Object} contains the values:

\begin{itemize}
  \item \texttt{FlowIndex\_Attribute} = 1
  \item \texttt{FlowType\_Attribute} = \texttt{UPDATE}
  \item \texttt{FlowFromIndex\_Attribute} = 2
  \item \texttt{FlowToIndex\_Attribute} = 1
\end{itemize}

The data flow \texttt{UPDATE} has thus been defined from the instance \texttt{SALESREP} to the instance \texttt{TBL\_SALESREP}. For internal use, this data flow definition has the index 1.

The data flow may be qualified further by introducing a new object. The object \texttt{CASE\_FlowDestination\_Object} defines the destination of the data flow. The destination of the data flow in the previous example was the database table \texttt{TBL\_SALESREP}. The object definition is:

\begin{itemize}
  \item \texttt{Object CASE\_FlowDestination\_Object(CASE\_FlowType\_Object)}
  \item \texttt{FlowDestinationIndex\_Attribute} of type Integer
  \item \texttt{FlowDestinationName\_Attribute} of type String
  \item \texttt{SetFlowDestinationIndex\_Method}
  \item \texttt{GetFlowDestinationIndex\_Method}
  \item \texttt{SetFlowDestinationName\_Method}
  \item \texttt{GetFlowDestinationName\_Method}
\end{itemize}

Thus, the flow of data from the \texttt{SALESREP} entity to the database table \texttt{TBL\_SALESREP} may be defined more precisely by adding a value for the \texttt{FlowDestinationName\_Attribute}:

\begin{itemize}
  \item \texttt{FlowDestinationName\_Attribute} = \texttt{DB TABLE TBL\_SALESREP}
  \item \texttt{FlowDestinationIndex\_Attribute} = 1
\end{itemize}
Once again, the value of the attribute `FlowDestinationIndex_Attribute` is for internal numbering.

With all prerequisites for the DFD object fulfilled, the object `CASE_DFD_Object` may be created. Since this object requires information from the two subclasses `CASE_Flow_Parent` and `CASE_EntityAttribute_Object`, multiple inheritance is once again used to create the object `CASE_DFD_Object`:

```
Object CASE_DFD_Object(CASE_Entity_Object, CASE_FlowDestination_Object)
```

All objects required have been defined. An inheritance diagram is now presented.
6.2.6 The diagram class hierarchy

It is common to show a schematic of any class created in an object-oriented environment. The following diagram indicates the complete structure of the diagram class. The diagram clarifies the inheritance from a higher level object to a lower level object.

![The Diagram Class Hierarchy](image-url)

Figure 6.4 - The diagram class hierarchy
6.3 Conclusion

This chapter has illustrated and defined the CASE tool component to be used for this study in an object-oriented sense. The advantages the object-oriented approach has for the CASE tool component may be summarised as being the following:

- the object-oriented structure enhances the modularity of the CASE tool component
- the diagrams defined by means of this CASE tool component can be manipulated as objects in their own right

The points mentioned facilitate the model developed in this study because the model itself is based on the object-oriented paradigm. By making use of a uniform approach, a general concept can be defined more easily. Furthermore, this chapter has defined a subset that can be used to illustrate the integration aspect of the proposed model in detail.
7.1 Introduction

As is stated by [DEVE91], to improve the quality and productivity of its computer products and services, a coherent group of tools and services is required by any organisation. The aim is to introduce a common culture to the creation of new software products. This culture is brought about by the creation and definition of concepts that can aid the organisation in its task. By adopting a software engineering methodology, such a culture can be more easily formed.

In the previous chapter, the CASE environment or component was defined in object-oriented terms. As was stated in the introductory paragraph of that chapter, it is not the aim of this study to redefine new CASE tools or new software engineering methodologies. Thus, a small but usable subset of a fictitious software engineering methodology will be defined. The subset will suffice to illustrate the functions and benefits of the feedback loop.

It is the aim of this chapter to define the software methodology component in an object-oriented manner.
7.2 Defining the methodology component

Traditionally, software engineering methodologies were never available in a computerised form. Developers made use of lists and books containing the steps of the methodology they had chosen to use for a particular development task.

As the concept of using a methodology became more and more important, computerised versions of many popular methodologies became available. As an example, a number of implementations of the methodology MERISE exist, each one being slightly different where model design and vocabulary are used [QUAN91].

In order for the integration between the methodology component and the CASE tool component defined in the previous chapter to be usable, the methodology component will cater for the following:

1. provision of tasks to request the identification of entities, attributes, relationships and data flows among entities
2. provision of tasks to request the creation of diagrams to include the entities, attributes, relationships and data flows identified by the user

These functions will now be discussed and defined.
7.3 Identifying the problem space

Whenever a developer or designer is faced with the prospect of creating a new software solution, the first step is to investigate and understand the problem space. [COAD90] The term problem space is used to represent the environment together with the proposed solution that must be investigated. Often, the developer is best advised to identify the components making up the problem space. These components are the entities, attributes, data flows and relationships among the entities.

[MITC90] states that the first development step is to produce a specification. Such a specification is normally an abstract model of the problem.

7.3.1 Identification of entities

The CASE tool component has made provision for the creation of entity-relationship diagrams and data flow diagrams. It is the aim of the software engineering methodology to offer tasks to complement this functionality of the CASE component.

The identification of entities is a natural starting point for any designer or developer required to create a new software solution. As is stated by [SNEE89], entities are the elements of the system or problem space being investigated. Examples of these elements are organisational units, applications, data areas and organisational resources. [COAD90] states that the primary motivation for identifying objects is to match the technical representation of a system more closely to the conceptual view of the real world.
These are the elements that have to be created or changed. All these entities have both fixed and variable attributes and characteristics. It is possible that the characteristics vary from one element to another. [SNEE89] Another name given to entities is the term object. [COAD90] The term object is used exactly as the concept object that was introduced earlier, thus supporting the object-oriented paradigm chosen for the model.

The identification of objects or entities within the problem space may be achieved by examining the problem space for structures, other systems, events remembered, roles played, locations and organisational units. [COAD90] As an example, the object SITE may be identified as an object once the developer considers the problem space he or she is working in. The problem space under investigation may be a software solution for a construction company - such a construction company may deal with various sites. As a further example, the object or entity SENSOR may be defined once the developer considers the problem space. If the problem space deals with a sensor that must interact with the system under consideration, then the object SENSOR should be identified. Such objects or entities may be represented diagrammatically as indicated by [COAD90]:

Figure 7.1 - SENSOR object example
7.3.2 Identification of attributes

Once entities or objects have been identified, the attributes for all objects must be found. The term attribute is once again used as was defined in a previous chapter. The way in which [OLLE88] defines the usage of attributes is analogous to the usage of attributes in this study - an attribute must have a name and every attribute defined for an entity has a unique name. Where the identification of attributes for entities in the software methodology is concerned, [OLLE88] states that each entity has at least one attribute. This is not to be confused with the definition of objects in the object-oriented programming sense for the components used for this study. By definition, an object in the object-oriented programming paradigm may have zero or more attributes associated with it.

The fixed and variable attributes of each object must be defined. [SNEE89] The fixed attributes are those that remain constant for most of the lifetime of an object. As an example, the location and sensor type attributes of the object SENSOR would be fixed attributes. An example of a variable attribute is the current state of the sensor. If the sensor reacts to temperature or pressure changes, for example, its state may be changing continually. Thus, the state of the sensor is variable since the value of this attribute changes whereas the sensors' location may be taken to be a fixed attribute since it is constant. In terms of defining the attributes, there is no difference between fixed and variable attributes.

It should be noted that the identification of variable attributes eases the identification of processes and actions that are to be carried out by the software solution proposed for the problem space.

As mentioned by [COAD90], special attention should be given to the identification of attributes. When identifying attributes for an object, it may
become apparent that a new object is required. In this case, the new object should be identified within the problem space if found to be required. As an example, consider the object OWNER which could have been identified within the problem space of a vehicle tracking system. The object could have the structure:

![ OWNER object with attributes ](image)

Figure 7.2 - OWNER object with attributes

The designer may feel that a new object is required to contain the information for the vehicle itself. Thus, the object VEHICLE is identified and the attributes pertaining to vehicles are moved to that object:

![ Adding a new object and moving attributes ](image)

Figure 7.3 - Adding a new object and moving attributes

In a similar manner, an object may be removed once the designer has seen that the attributes it contains should be moved to another object. This is the case specifically if an object has only one attribute associated with it.

As an example of this, consider the following objects:
It makes more sense to move the attribute Address from the object LOCATION to the object OWNER. In doing so, the object LOCATION becomes redundant since it no longer contains any information. This technique of moving attributes and then removing certain objects is useful to control complexity. [COAD90] The resultant object is an extended OWNER object:

In the following section, relationships are briefly discussed.
7.3.3 Identification of relationships

Relationships pertain specifically to entity-relationship diagrams. This was illustrated in the previous chapter, where an example of an entity-relationship diagram was given.

The relationships that exist between two entities or objects are the following:

- one-to-one relationships
- one-to-many relationships
- many-to-many relationships

As stated by [OLLE88], the designer of an information system will need to designate how the entities are interrelated. Normally, one entity is associated in some way with at least one other entity. Occasionally, it is necessary to define a recursive relationship for an entity.

As an example of a one-to-one relationship, consider the objects VEHICLE and REGISTRATION. Any particular vehicle may be registered once only. Conversely, any particular registration pertains only to one vehicle. One-to-one relationships are usually indicated by the abbreviation 1-1.

As an example of a one-to-many relationship, consider the example introduced in the previous chapter. In that example, the entities introduced were CUSTOMER and SALESREP. Any particular customer is looked after by one
sales representative only. However, any given sales representative can have one or more customers to attend to. One-to-many relationships are usually indicated by the abbreviation 1-N.

The last example concerns the many-to-many relationship. If the objects identified by the designer are AIRCRAFT and ROUTE, then a possible many-to-many relationship may exist. In this case, a particular aircraft could be used for any route. In the same way, any given route could be flown by one or more aircraft. Many-to-many relationships are usually indicated by the abbreviation N-N.
7.3.4 Identification of data flows

Data flows are of particular importance where data flow diagrams are concerned. The previous chapter included an example of such a diagram. Data flows determine the flow of information between various objects identified within the problem space. Data flow diagrams may also be called information flow diagrams. [MACR87]

It is usual for data flow diagrams to be used as a means of identifying the source and destination points of information flows. As such, data flows are independent of timing constraints. [MACR87] This means that all data flows can be regarded as all working together, even though some of them may work only at specific times or once a certain action occurs.

It is furthermore possible that certain tasks are interdependent or may operate in parallel. In other words, if some task has almost been completed, information already gathered by that task can be used to start or continue with another related task. The advantage of such parallel tasks is the speed and efficiency with which the work can be completed.

As an example of such an interdependency and parallel tasks, consider the following example. Assume that a certain methodology prescribes that both a data flow diagram and an entity-relationship diagram must be constructed. Since results and findings from the ERD are used in the DFD, it makes sense to state that these tasks are interdependent. Furthermore, at any one step during the construction of the ERD, construction of the DFD may begin, given that enough information is available.
CHAPTER 7: DEFINING THE SOFTWARE METHODOLOGY ENVIRONMENT

The diagram below illustrates this principle:

![Diagram of parallel activities with constructs ERD and DFD]

Figure 7.9 - Parallel activities

It must be remembered that parallel tasks must be synchronised. In other words, a change to information in the ERD will impact the DFD. The model developed in this study attempts to solve this problem by way of the feedback loop.

The data flows from some source object and may be influenced by certain processing activities. Once processed, the information is stored in a destination object. As an example, the objects SCREEN_HANDLER and SCREEN may have been identified in the problem space. The data flow is from the SCREEN_HANDLER to the SCREEN. The type of data flow may be described as a message display flow. The information being passed to the screen is a message to be displayed there.

![Diagram of message display data flow]

7.10 - Display message data flow

The most common type of flow that is used in information system design is the flow of information to some storage medium, such as a file or a database table. As an example, consider the storage of an employee record in a suitable table in...
a database. The objects under consideration are EMPLOYEE and TBL_EMPLOYEE. The data flow is one employee record, for example.

![Diagram of data flow](image)

**Figure 7.11 - Save employee record data flow**

### 7.4 Deriving the methodology object class

The examples provided during the discussion of the methodology give some insight into the requirements necessary for the methodology object class. The starting point for the object class definition is an empty object, the *Method_Parent*. This object is the uppermost in the methodology object class. The structure of the object is as follows:

```
Object Method_Parent
```

The methodology component requires information regarding a number of components. There will be a project information component and a methodology information component. All the subcomponents required for the methodology will also be developed.
7.4.1 Deriving the project information system object class

The methodology requires information regarding the project being undertaken as well as other administrative information. It is common to make use of project planning tools and systems to keep track of progress. The parent of the project information system class is the Method_Project_Parent.

\[ \text{Object Method\_Project\_Parent}(\text{Method\_Parent}) \]

The information regarding the project will be stored in the object Method_ProjectInfo\_Object. Please note that the same conventions as introduced in the previous chapter will be used. Thus, to store information in an attribute of any object, a Set method will be provided. Similarly, a Get method must be present for the information to be retrieved.

\[ \text{Object Method\_ProjectInfo\_Object}(\text{Method\_Project\_Parent}) \]
\[ \text{ProjectName\_Attribute} \text{ of type String} \]
\[ \text{ProjectStartDate\_Attribute} \text{ of type String} \]
\[ \text{ProjectEndDate\_Attribute} \text{ of type String} \]
\[ \text{ProjectLeader\_Attribute} \text{ of type String} \]
\[ \text{SetProjectName\_Method} \]
\[ \text{GetProjectName\_Method} \]
\[ \text{SetProjectStartDate\_Method} \]
\[ \text{GetProjectStartDate\_Method} \]
\[ \text{SetProjectEndDate\_Method} \]
\[ \text{GetProjectEndDate\_Method} \]
\[ \text{SetProjectLeader\_Method} \]
\[ \text{GetProjectLeader\_Method} \]
The information stored within this object is sufficient to represent the most important information required for a particular project.

Every project consists of a number of tasks. The tasks are arranged in order of their planned execution together with a duration and a start and a finish date for that particular task. The concept for linking objects by way of index numbers introduced in the previous chapter will be used extensively in this chapter.

The object to contain one project task together with its related information is the object `Method_ProjectTask_Object`. Note that there is a reference to an index in the methodology object. This allows the project manager to link tasks directly to the specific steps given by the methodology.

```
Object Method_ProjectTask_Object(Method_ProjectInfo_Object)
  ProjectTaskIndex_Attribute of type Integer
  TaskParentIndex_Attribute of type Integer
  MethodPhaseIndex_Attribute of type Integer
  TaskDescription_Attribute of type String
  TaskDuration_Attribute of type String
  TaskStartDate_Attribute of type String
  TaskEndDate_Attribute of type String
  Set_ProjectTaskIndex_Method
  Get_ProjectTaskIndex_Method
  Set_TaskParentIndex_Method
  Get_TaskParentIndex_Method
  Set_MethodPhaseIndex_Method
  Get_MethodPhaseIndex_Method
  Set_TaskDescription_Method
  Get_TaskDescription_Method
  Set_TaskDuration_Method
```
An explanation of this object is in order. As was stated, the index entries are used to link one instance of this object to another instance of this object. Assume that two tasks have been defined for the project.

The tasks that have been defined for this example have the indexes 1 and 2. The instance details for the first instance are:

\[
\begin{align*}
ProjectTaskIndex\_Attribute &= 1 \\
TaskParentIndex\_Attribute &= 0 \\
MethodPhaseIndex\_Attribute &= 0 \\
TaskDescription\_Attribute &= \text{Investigate user requirements} \\
TaskDuration\_Attribute &= 3 \text{ days} \\
TaskStartDate\_Attribute &= 1/1/96 \\
TaskEndDate\_Attribute &= 4/1/96
\end{align*}
\]

For the second instance the details are:

\[
\begin{align*}
ProjectTaskIndex\_Attribute &= 2 \\
TaskParentIndex\_Attribute &= 1 \\
MethodPhaseIndex\_Attribute &= 0 \\
TaskDescription\_Attribute &= \text{Load operating system for mainframe} \\
TaskDuration\_Attribute &= 5 \text{ days} \\
TaskStartDate\_Attribute &= 4/1/96 \\
TaskEndDate\_Attribute &= 10/1/96
\end{align*}
\]
The link from the task with index 2 to the task with index 1 is accomplished by the entry for the attribute TaskParentIndex_Attribute. Sequential numbering of tasks is preferable, even though any numbering scheme is permissible. Notice that the contents for the attribute MethodPhaseIndex_Attribute is 0. The convention used here will require a 0 to be stored in that attribute whenever a project task does not relate to an activity required in a phase of the methodology. Should a phase in the methodology directly relate to a task in the project plan, the index for that phase of the methodology should appear in this attribute.

Since the project information required for purposes of this study consists only of the type of information shown above, the project information class has been fully defined. By using the link attributes, any number of tasks may be defined for a project.

7.4.2 Deriving the methodology object class

It has been mentioned that a small subset of a methodology will be defined for purposes of this study. Since a methodology consists of a number of sequentially executable tasks, the structure of the methodology object class will be very similar to the object class defined for the project information system class.

The parent object in the methodology object class is the object Method_Method_Parent.

Object Method_Method_Parent(Method_Parent)
CHAPTER 7: DEFINING THE SOFTWARE METHODOLOGY ENVIRONMENT

Information is associated with the methodology. This information is stored in an instance of the following object definition:

```
Object Method_MethodInfo_Object(Method_Method_Parent)
    MethodName_Attribute of type String
    MethodVersion_Attribute of type String
    SetMethodName_Method
    GetMethodName_Method
    SetMethodVersion_Method
    GetMethodVersion_Method
```

The information required for the methodology component is minimal. A version identification is required since a revision to a particular methodology may cause inconsistencies when used with data from a different version. The name attribute allows the methodology to be formally named.

As for the project information object class, a number of links will be created to define the phases of the methodology. The individual phases of the methodology are represented by instances of the following object definition:

```
Object Method_MethodPhase_Object(Method_MethodInfo_Object)
    PhaseIndex_Attribute of type Integer
    PhaseParentIndex_Attribute of type Integer
    PhaseDescription_Attribute of type String
    PhaseComplete_Attribute of type Boolean
    Set_PhaseIndex_Method
    Get_PhaseIndex_Method
    Set_PhaseParentIndex_Attribute
    Get_PhaseParentIndex_Attribute
    Set_PhaseDescription_Method
```
As an example, consider the following phases of a methodology named MethodStep:

1. identify all entities within the problem space
2. determine attributes for all the entities identified
3. determine the relationships between the entities identified

To represent these phases in terms of the object class defined above, a number of instances of objects are required. The first is one instance of the object Method_MethodInfo_Object with the following attribute values:

- MethodName_Attribute = MethodStep
- MethodVersion_Attribute = 1.0

In order to represent the phases defined within the methodology, three instances of the object definition Method_MethodPhase_Object are required. Each instance links to its parent phase by way of the PhaseParentIndex_Attribute value.

- PhaseIndex_Attribute = 1
- PhaseParentIndex_Attribute = 0
- PhaseDescription_Attribute = Identify all entities

- PhaseIndex_Attribute = 2
- PhaseParentIndex_Attribute = 1
- PhaseDescription_Attribute = Identify all attributes for all entities
Phaselndex_Attribute = 3
PhaseParentlndex_Attribute = 2
PhaseDescription_Attribute = Identify all relationships

The index ensures that the phases follow on one another in the correct sequence. As was mentioned previously, activities or phases may be executed in parallel. In this case, it is once again important that changes found in a certain phase are filtered through to the other phases.

Also, the index for a phase may be used by the project information system to refer directly to a particular phase in the methodology. This link is useful for project managers who are not familiar with the individual steps of the methodology. By linking to the phases of the methodology, the project manager ensures that no important steps are forgotten.

The description of the phase being executed is for the user, who is prompted to enter the information interactively. The question of where the information regarding the entities, attributes and relationships is to be stored must be answered. A number of special object definitions must be created for this purpose. The object definitions will be presented in the next sections.

7.4.3 Deriving the entity object class

Entities have been identified as being very important for both diagram types supported by the CASE tool component in the previous chapter. These diagram types are the data flow diagram and the entity-relationship diagram. The object definitions presented here are similar to the definitions presented in the previous chapter, but should not be assumed to fulfil the same function. The object definitions made here are used exclusively by the methodology component.
A parent object definition is created for the entity object class, as is customary:

\[ \text{Object Method\_Entity\_Parent}(\text{Method\_Parent}) \]

The entity details are stored in instances of the object definition \textit{Method\_Entity\_Object}:

\[ \text{Object Method\_Entity\_Object}(\text{Method\_Entity\_Parent}) \]

\begin{itemize}
  \item \textit{EntityIndex\_Attribute} of type \texttt{Integer}
  \item \textit{EntityName\_Attribute} of type \texttt{String}
  \item \texttt{SetEntityIndex\_Method}
  \item \texttt{GetEntityIndex\_Method}
  \item \texttt{SetEntityName\_Method}
  \item \texttt{GetEntityName\_Method}
\end{itemize}

The index is used for purposes of linking a number of entities to their attributes. For the entity \textit{OWNER}, the following instance of \textit{Method\_Entity\_Object} is required:

\begin{itemize}
  \item \textit{EntityIndex\_Attribute} = 1
  \item \textit{EntityName\_Attribute} = \texttt{OWNER}
\end{itemize}
CHAPTER 7: DEFINING THE SOFTWARE METHODOLOGY ENVIRONMENT

7.4.4 Deriving the entity-attribute object class

As attributes are associated with entities, a new object definition must be made to cater for the storage of attributes. This is the `Method_EntityAttribute_Object` object definition:

```
Object Method_EntityAttribute_Object(Method_Entity_Object)

  EntityAttributeIndex_Attribute of type Integer
  EntityAttributeName_Attribute of type String
  EntityParentIndex_Attribute of type Integer
  SetEntityAttributeIndex_Method
  GetEntityAttributeIndex_Method
  SetEntityAttributeName_Method
  GetEntityAttributeName_Method
  SetEntityParentIndex_Method
  GetEntityParentIndex_Method
```

The attribute NAME for the entity OWNER above would be represented by the instance of `Method_EntityAttribute_Object`:

```
EntityAttributeIndex_Attribute = 1
EntityAttributeName_Attribute = NAME
EntityParentIndex_Attribute = 1
```
7.4.5 Deriving the relationship object class

Entities are involved in relationships. The object definition class to represent relationships is started off with a parent object:

\[ \text{Object Method\_Relationship\_Parent(} \text{Method\_Parent)} \]

To represent relationships, the indexes for each entity are used. The object definition \text{Method\_Relationship\_Object} is capable of representing relationships:

\[ \text{Object Method\_Relationship\_Object(} \text{Method\_Relationship\_Parent)} \]

\begin{itemize}
  \item \text{RelationshipIndex\_Attribute} of type Integer
  \item \text{RelationshipName\_Attribute} of type String
  \item \text{FromEntityIndex\_Attribute} of type Integer
  \item \text{ToEntityIndex\_Attribute} of type Integer
  \item \text{RelationshipType\_Attribute} of type String
  \item \text{SetRelationshipIndex\_Method}
  \item \text{GetRelationshipIndex\_Method}
  \item \text{SetRelationshipName\_Method}
  \item \text{GetRelationshipName\_Method}
  \item \text{SetFromEntityIndex\_Method}
  \item \text{GetFromEntityIndex\_Method}
  \item \text{SetToEntityIndex\_Method}
  \item \text{GetToEntityIndex\_Method}
  \item \text{SetRelationshipType\_Method}
  \item \text{GetRelationshipType\_Method}
\end{itemize}
To represent the many-to-many relationship between the entities AIRCRAFT and ROUTE as presented earlier, two instances of Method_Entity_Object are required:

\[
\begin{align*}
  \text{EntityIndex\_Attribute} &= 1 \\
  \text{EntityName\_Attribute} &= \text{AIRCRAFT} \\
  \\
  \text{EntityIndex\_Attribute} &= 2 \\
  \text{EntityName\_Attribute} &= \text{ROUTE}
\end{align*}
\]

To create a N-N relationship between these entities, one instance of Method_Relationship_Object is required:

\[
\begin{align*}
  \text{RelationshipIndex\_Attribute} &= 1 \\
  \text{RelationshipName\_Attribute} &= \text{AIRCRAFT-ROUTE} \\
  \text{FromEntityIndex\_Attribute} &= 1 \\
  \text{ToEntityIndex\_Attribute} &= 2 \\
  \text{RelationshipType\_Attribute} &= \text{N-N}
\end{align*}
\]
7.4.6 Deriving the data flow object class

Entities are also the primary component involved in data flows. Data flows from one entity, the source, to another entity, the destination. To start the data flow class definition off, a parent object is defined:

```
Object Method_Flow_Parent(Method_Parent)
```

Actual data flows are represented in a similar way to relationships. The structure that will be used is defined as follows:

```
Object Method_Flow_OBJECT(Method_Flow_Parent)
  FlowIndex_Attribute of type Integer
  FlowName_Attribute of type String
  FromEntityIndex_Attribute of type Integer
  ToEntityIndex_Attribute of type Integer
  FlowType_Attribute of type String
  SetFlowIndex_Method
  GetFlowIndex_Method
  SetFlowName_Method
  GetFlowName_Method
  SetFromEntityIndex_Method
  GetFromEntityIndex_Method
  SetToEntityIndex_Method
  GetToEntityIndex_Method
  SetFlowType_Method
  GetFlowType_Method
```
An example should clarify this structure. Assume the entities SCREEN and SCREEN_HANDLER exist as presented earlier. Two instances of Method_Entity_Object are required:

\[
\begin{align*}
\text{EntityIndex\_Attribute} & = 1 \\
\text{EntityName\_Attribute} & = \text{SCREEN} \\
\text{EntityIndex\_Attribute} & = 2 \\
\text{EntityName\_Attribute} & = \text{SCREEN\_HANDLER}
\end{align*}
\]

To represent the data flow "display message" between these entities, one instance of Method_Flow_Object is required:

\[
\begin{align*}
\text{FlowIndex\_Attribute} & = 1 \\
\text{FlowName\_Attribute} & = \text{Display message} \\
\text{FromEntityIndex\_Attribute} & = 2 \\
\text{ToEntityIndex\_Attribute} & = 1 \\
\text{FlowType\_Attribute} & = \text{DISPLAY MESSAGE}
\end{align*}
\]

All prerequisites for the objects required by the methodology component in this study have been defined and described. The class hierarchy diagram will present an overview of all object definitions made in this chapter.
7.4.7 The methodology class hierarchy

The diagram presented here shows all object classes as defined in this chapter. The inheritance aspects are summarised by the diagram.

![The Methodology Class Hierarchy](image)

Figure 7.12 - The methodology class hierarchy

7.5 Conclusion

This chapter has illustrated and defined the methodology component to be used for this study in an object-oriented sense. The advantages concerning the object-oriented approach for the CASE tool component are also applicable to the methodology component. The advantages are:

- the object-oriented structure enhances the modularity of the methodology component
- the phases of the chosen methodology may be manipulated as objects in their own right
- the instances as defined by the object class definitions are able to interact with the CASE tool component as defined in the previous chapter
CHAPTER 7: DEFINING THE SOFTWARE METHODOLOGY ENVIRONMENT

The points mentioned facilitate the model developed in this study because the model itself is based on the object-oriented paradigm. By making use of a uniform approach, a general concept can be defined more easily. Furthermore, this chapter has defined a subset that can be used to illustrate the integration aspect of the proposed model in detail.

In the next chapter, a short discussion of the repository is presented. The aim of chapter 8 is to introduce and define the necessary repository structures, such as database tables and triggers, that are necessary for the feedback loop's operation.
8.1 Introduction

In the previous chapters, the concept of a repository has been mentioned a number of times. This was the case specifically for chapters 6 and 7 where the components of the model were developed in detail. The repository is important for the storage of information that will enable the feedback loop to do its work. The repository can provide a number of useful structures that enable the feedback loop to operate more efficiently. This chapter will introduce those components and will define them in more detail.

8.2 Useful repository structures

The structures used to store information for reference by the feedback loop will be database tables. The necessary tables will be closely related to the object definitions presented in chapters 6 and 7. The structures aiding the feedback loop's operation will be database triggers, or stored procedures. These triggers will act upon information contained within those tables.

It will be seen that the repository required for the model in this study requires additional structures. The relational database model as defined by Dr. E.F. Codd does not directly cater for the stored procedure or trigger functionality, for example. Since these functions are not available in generic relational database management systems, the extended relational system will be investigated.
8.3 Extended relational systems

The extended relational database has come about by extending the relational DBMS with a number of additional functions. [BERT93] The architecture of extended relational systems represents database systems that provide new functionality through new or extended query languages that incorporate procedures and other object data management system (ODMS) features. [CATT91] It is usual that extended database systems present the programmer with two separate environments, namely the application programming language and the extended database query language.

From this short description of extended relational systems it should be clear that a number of tasks can be completed more easily and more efficiently than under purely relational systems. The concept of object data management is an attractive feature for the model being developed in this study.

8.3.1 Object data management

Object data management is a subject that has not been fully investigated. [CATT91] As stated by [CATT91], object data management systems include both object and data management capabilities. A very important statement is made by [HARE94]: on a recent assignment, a relational database management system providing triggers, procedures and rules, was proved to be capable of supporting object-oriented information modelling facilities. It is this feature of the extended relational systems that is of the utmost importance for the model developed in this study. The reason for this should be clear: the model makes use of the object-oriented paradigm.
Object data management systems (ODMS) have been implemented using a wide variety of data models. The relational model has, as mentioned above, been implemented for an ODMS. The features of an ODMS are listed by [CATT91]:

a) provides features of objects, including object identifiers and attributes
b) provides relationships, reference attributes and sets
c) provides procedures (methods) and encapsulation
d) provides object types, classes and hierarchies
e) concurrency control and recovery

These are important aspects for the model under development, particularly because the model is object based. A discussion of each one is now presented.

a) Provides features of objects

The model in this study is based on the object-oriented paradigm. The support provided by an ODMS using a relational data model allows object grouping. [CATT91] This means that objects can serve to group data that pertain to one real-world entity. By employing the relational structure as a base for this, objects may be grouped by creating records in relations. Just as important as object grouping is the capability of identifying objects.

An object is identified by the value it contains. Such a structure already exists in the relational model - the concept of a primary key. [CATT91] states that primary keys demand that application programmers either generate their own unique identifiers or use human-meaningful names as identifiers. Modern database management systems provide SQL extensions to provide this functionality - thus, a primary key for a new row in a table is generated automatically by the database without any intervention. The problem identified by [CATT91] is thus solved.
Attributes are identified by [CATT91] as a further requirement of an ODMS. As stated by [CATT91], functions mapping objects to simple values may be stored within the database. Such functionality is provided by most modern database management systems. The stored procedures are stored directly within the database and may be executed in a variety of ways - triggers are examples of stored procedures that are executed once a certain condition is met.

b) Provides relationships

Relationships among objects may be created by class hierarchies or by values contained within objects. In the relational data model, references are created by making use of foreign and primary key values. Relationships in the model developed in this study are not important for objects - objects will be created as required and will represent one real-world entity only. Thus, relationships among the objects are created by instantiating objects in various levels of the class hierarchy.

Relationships are not important for implementation of the model, except where normal references are required from attributes to other attributes. This is the situation whenever primary and foreign keys are implemented.

c) Provides procedures

Database triggers are functions associated with extended relational systems. The trigger is a procedure that is created to perform some action. Usually, the activation of such a procedure is dependant on some occurrence, such as the insertion of a new row in a table or another condition that is fulfilled. It will be shown later that the trigger is an important building block for the model. Triggers will be used to convey information to the feedback loop.
In general, it can be stated that an ODMS provides a fully functional programming language for the definition of methods. [CATT91] It is usual for the SQL language to be extended to provide variables, for-loops and procedures. Thus, the programmer can perform computation in the database language instead of the application language. [CATT91]

The capability of creating procedures is analogous to the creation of methods. Methods behave in the same way that was described in a previous chapter. The passing of messages among objects is the primary function of methods.

Trigger or stored procedure definitions will be presented as follows:

```
Trigger_TriggerName
for <Table> on <Action> as
(<Trigger_Definition>);
```

The trigger has a descriptive name. By convention, this name will have the prefix "Trigger". The trigger is executed whenever a certain event occurs for a certain database table. Thus, the name of the table associated with the trigger must be specified. To determine when the trigger should execute, an action must be specified. Valid actions are INSERT, UPDATE and DELETE. Whenever the specified action occurs for the specified table, the trigger is executed. The actual functionality provided by the trigger is specified by way of an SQL statement.

An example should clarify the process of trigger definitions. Assume two tables, TBL_TABLE1 and TBL_TABLE2, exist within the database. Whenever a new record is inserted into TBL_TABLE1, the contents of the newly inserted fields FLD_FIELD1 and FLD_FIELD2 are also to be inserted into TBL_TABLE2. The
value of FLD_FIELD1 is to be transferred to FLD_F1 in TBL_TABLE2, the value of FLD_FIELD2 is to be transferred to FDL_F2 in TBL_TABLE2. The resulting trigger definition is:

```
Trigger_InsertIntoTable2
  for TBL_TABLE1 on INSERT
  as
  (INSERT (FLD_FIELD1, FLD_FIELD2)
  (FLD_F1, FLD_F2)
  INTO TBL_TABLE2);
```

Trigger definitions for the action UPDATE to a table are slightly different. These triggers have the form:

```
Trigger_TriggerName
  for <Table> on UPDATE TO <FieldList> as
  (<Trigger_Definition>);
```

The difference is the inclusion of a mandatory list of fields that are affected by the update function. If the trigger applies to more than one field, the fields are separated by commas. The trigger is executed whenever one of the fields in the list is updated. This allows the definition of very specialised triggers.

The repository monitors the actions and calls the necessary triggers automatically. It is possible to create more than one trigger for the same event, the only requirement is that each trigger's name is unique.
d) Provides object types and hierarchies

The availability of object types and class hierarchies is no different from that of an object-oriented system. Thus, the ODMS may contain class hierarchies and objects as instances of those classes. Representing an ODMS in a relational system means that tables must be considered to be types. This is supported by [CATT91] who states that in the relational model, tables may be considered to be types. More precisely, the definition of a table's attributes are the type definition, and the rows of a table are the instances of the type.

As was stated previously, inheritance may also be called generalisation. [HARM91] Thus, a number of entries in a particular relation or table may be generalised by saying that they belong to a certain class. This is generalisation and is important for the creation of structures to form hierarchies and ultimately to provide a useful structure for inheritance by new structures.

e) Provides concurrency control and recovery

All modern relational database management systems provide concurrency and recovery controls. Thus, a number of active processes may use resources within the database concurrently, without damaging any of the structures or data within the database. In a similar fashion, recovery procedures exist to recover information in the case of an error or failure situation. Whereas relational database systems provide an adequate concurrency mechanism by way of the transaction principle, object data management systems require certain modifications to ensure that concurrency control exists.

As stated by [CATT91], concurrency, recovery and physical and logical integrity are important because data within the DBMS are maintained once the user session or the application program terminate. It will be assumed that the
CHAPTER 8: DEFINING REPOSITORY STRUCTURES

repository used for the model in this study ensures the consistency of information stored within its tables. Furthermore, should a transaction fail for any reason, the entire transaction will be undone. In all cases, transactions provide the atomic unit of control that is required whenever recovery or concurrency is needed. [DATE90] Transactions play an important part in the storage of data within a DBMS or ODMS.

8.4 Repository structures for the CASE and methodology components

The database tables and triggers will be discussed for both components involved. Definitions for the CASE component will be presented first, followed by the definitions for the methodology component. The reader may find it useful to refer back to the object definitions as presented in chapter 6 for the CASE component and chapter 7 for the methodology component.

8.4.1 Repository structures for the CASE component

The table definitions for the CASE tool component will be defined first. Please refer to the object definitions in chapter 6 if required. The table definitions are not different from the object definitions given. Generally it may be stated that all the attributes given within an object constitute a table within the database. Additional fields have been added only in a few exceptions. In other cases, link tables exist for lookup purposes.
The table TBL_CASE_DIAGRAMDETAIL stores information for instances of the objects CASE_DiagramDetail_Object and CASE_DiagramType_Object.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_DIAGRAM_VERSION</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_DATE_MODIFIED</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_CREATED_BY</td>
<td>String</td>
</tr>
<tr>
<td>FLD_FK_DIAGRAM_TYPE</td>
<td>Integer</td>
</tr>
</tbody>
</table>

The fields in this table correspond directly to the object definition. For example, the field FLD_CREATED_BY contains the name of the designer who created the diagram. The field FLD_ENTRY_INDEX is automatically updated whenever a new row is inserted. Thus, if a third row is added to the table, the entry for FLD_ENTRY_INDEX will be 3. This concept is used for all the tables in the repository and ensures that every row in any table has a unique primary key.

The table TBL_CASE_DIAGRAMTYPE is a link or lookup table for the table described above. The field FLD_FK_DIAGRAM_TYPE links to diagram details in this table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_DIAGRAM_TYPE</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_DESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Information pertaining to the various diagram types is stored in this table. Every diagram type is allocated a number. The number allocated to a certain diagram type may be used to reference the general description for that diagram.
CHAPTER 8: DEFINING REPOSITORY STRUCTURES

For instances of the object `CASE_Entity_Object` the table `TBL_CASE_ENTITY` contains the relevant information. This table is primarily a lookup table for entity names and descriptions.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ENTITY_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_DESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Every entity identified by the designer will have one entry in this table. The entity name and a short description for that entity may be stored in this table.

The table `TBL_CASE_ENTITY_ATTRIBUTE` contains information for instances of the object `CASE_EntityAttribute_Object`. In order to find entity details for any attribute, the foreign key `FLD_FK_ENTITY_INDEX` refers back to the previous table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_ATTRIBUTE_INDEX</td>
<td>Integer</td>
</tr>
</tbody>
</table>

In order to associate attributes with entities, this table contains mostly foreign key values. Thus, the function of this table is to associate all attributes belonging to one entity to that entity.

Just as there was a lookup table for entities, so there is a lookup table for attributes. This table is called `TBL_CASE_ATTRIBUTE`. In the previous table, attribute details can be retrieved by searching for the value of the foreign key field `FLD_FK_ATTRIBUTE_INDEX` in this table.
CHAPTER 8: DEFINING REPOSITORY STRUCTURES

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ATTRIBUTE_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ATTRIBUTE_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_DESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Every attribute identified by the designer will have one entry in this table. The attribute name and a short description for that attribute may be stored in this table.

Instances of the two objects `CASE_Relationship_Object` and `CASE_RelationshipType_Object` retrieve and store information in the table `TBL_CASE_RELATIONSHIP`. The entries in this table are mostly foreign key references to the entity lookup table and the relationship type lookup table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_FROM_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_TO_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_RELATIONSHIP_INDEX</td>
<td>Integer</td>
</tr>
</tbody>
</table>

In order to associate relationships with entities, this table contains mostly foreign key values. Thus, the function of this table is to associate all relationships from one entity to another.

---

Page 139
CHAPTER 8: DEFINING REPOSITORY STRUCTURES

TBL_CASE_RELATIONSHIP_TYPE is a lookup table for relationship types. References to the values in this table exist from the previously mentioned table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_RELATIONSHIP_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_RELATIONSHIP_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_DESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Every relationship type identified by the designer will have one entry in this table. The relationship name and a short description for that relationship type may be stored in this table.

To represent data flows, instances of the objects CASE_FlowType_Object and CASE_FlowDestination_Object store information in the table TBL_CASE_FLOW. Once again, this table contains mostly foreign key references.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_FROM_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_TO_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_FLOW_INDEX</td>
<td>Integer</td>
</tr>
</tbody>
</table>

In order to associate data flows with entities, this table contains mostly foreign key values. Thus, the function of this table is to associate all data flows from one entity to another.
CHAPTER 8: DEFINING REPOSITORY STRUCTURES

TBL_CASE_FLOTYPE is a lookup table for data flow types. References to the values in this table exist from the previously mentioned table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FLOW_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FLOW_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLDDESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Every data flow identified by the designer will have one entry in this table. The data flow name and a short description for that data flow may be stored in this table.

It is assumed that the information written into the CASE components' tables is written once-off. In other words, all the information pertaining to a new diagram is written to the tables at one particular point in time. Even though the method of storing information depends entirely on the internal operation of a CASE tool, the CASE tool in this study is assumed to store its information in this way.

Given that this is true, one or more triggers must exist in the database to inform the feedback loop of new information having been entered by a user. Even though the feedback loop is discussed in detail in the next chapter, a trigger definition is presented here for the purpose of informing the feedback loop of new information having been introduced. For now, assume that the feedback loop operates with the aid of control tables in the repository. The main control table has the name TBL_FEEDBACK_CONTROL.
A suitable trigger definition, presented in basic pseudocode, to inform the feedback loop of a change in the information contained in the CASE tool could be as follows:

```
Trigger TriggerCASEUpdate
    for <List_of_CASE_Tables>
    on INSERT
    as
    INSERT
    <Control_Information>
    into TBL_FEEDBACK_CONTROL;
```

The trigger executes whenever information is inserted into the CASE tool component's database tables. This is indicated by `<List_of_CASE_Tables>`. The action taken by the trigger is to create a request for integration in the control table TBL_FEEDBACK_CONTROL. The integration request is indicated by `<Control_Information>` and will be explained in greater detail in the next chapter.

### 8.4.2 Repository structures for the methodology component

The table definitions for the methodology component are defined next. Please refer to the object definitions in chapter 7 if required. The table definitions are generally not different from the object definitions given. It may be stated that all the attributes given within an object constitute a table within the database. In a few exceptions additional fields have been added. In other cases, link tables exist for lookup purposes.

Wherever required, trigger definitions will be presented.
Instances of Method_ProjectInfo_Object store and retrieve general project information in the database table TBL_METHOD_PROJECTINFO.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_PROJECT_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_PROJECT</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_START_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_END_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_PROJECT_LEADER</td>
<td>String</td>
</tr>
</tbody>
</table>

The fields in this table correspond directly to the object definition. For example, the field FLD_PROJECT_LEADER contains the name of the project manager.

Individual project tasks and subtasks are stored in the database table TBL_METHOD_PROJECTTASK. Method_ProjectTask_Object instances make use of this table. The link to the project is maintained by the foreign key field FLD_FK_PROJECT_ID to the previously mentioned table. To indicate that one task follows another, the previous tasks number is stored in the field FLD_PARENT_TASK.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_PROJECT_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_TASK_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_PARENT_TASK</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_METHODPHASE_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_TASK_DESCRIPTION</td>
<td>String</td>
</tr>
<tr>
<td>FLD_START_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_END_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_DURATION</td>
<td>Integer</td>
</tr>
</tbody>
</table>
CHAPTER 8: DEFINING REPOSITORY STRUCTURES

All the global project information is stored in this table. The individual tasks are linked to one another by way of internal reference fields. In other words, any one table entry contains links to another table entry within this table.

Information stored for the actual methodology is very similar to that stored for the project. TBL_METHOD_METHODINFO contains this information. Method_MethodInfo_Object instances access this table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_METHOD_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_METHOD_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_METHOD_DESCRIPTION</td>
<td>String</td>
</tr>
<tr>
<td>FLD_METHOD_VERSION</td>
<td>String</td>
</tr>
</tbody>
</table>

Information pertaining to the methodology in use is stored within this table. As an example, the field FLD_METHOD_NAME contains the name of the methodology in use by the designer.

Method_MethodPhase_Object instances use the table TBL_METHOD_METHODPHASE for information storage. This table is very similar to the table containing the individual project tasks.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_PHASE_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_METHOD_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_PHASE_DESCRIPTION</td>
<td>String</td>
</tr>
<tr>
<td>FLD_PHASE_COMPLETE</td>
<td>Boolean</td>
</tr>
<tr>
<td>FLD_TIME_COMPLETE</td>
<td>Time</td>
</tr>
<tr>
<td>FLD_PHASE_PARENT</td>
<td>Integer</td>
</tr>
</tbody>
</table>
All the global methodology information is stored in this table. The individual phases are linked to one another by way of internal reference fields. In other words, any one table entry contains links to another table entry within this table.

The storage for entities, attributes, data flows and relationships is virtually identical to the storage of these objects in the CASE component. TBL_METHOD_ENTITY for instances for object Method_Entity_Object is used.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ENTITY_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_DESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Every entity identified by the designer will have one entry in this table. The entity name and a short description for that entity may be stored in this table.

The table TBL_METHOD_ENTITYATTRIBUTE contains information for instances of the object Method_EntityAttribute_Object. In order to find entity details for any attribute, the foreign key FLD FK ENTITY_INDEX refers back to the previous table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD FK ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD FK_ATTRIBUTE_INDEX</td>
<td>Integer</td>
</tr>
</tbody>
</table>

In order to associate attributes with entities, this table contains mostly foreign key values. Thus, the function of this table is to associate all attributes belonging to one entity to that entity.

Instances of the object Method_Relationship_Object retrieve and store information in the table TBL_METHOD_RELATIONSHIP. The entries in this
table are mostly foreign key references to the entity lookup table and the relationship type lookup table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_FROM_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_TO_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_RELATIONSHIP_INDEX</td>
<td>Integer</td>
</tr>
</tbody>
</table>

In order to associate relationships with entities, this table contains mostly foreign key values. Thus, the function of this table is to associate all relationships from one entity to another.

**TBL_METHOD_RELATIONSHIPTYPE** is a lookup table for relationship types. References to the values in this table exist from the previously mentioned table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_RELATIONSHIP_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_RELATIONSHIP_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_DESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Every relationship type identified by the designer will have one entry in this table. The relationship name and a short description for that relationship type may be stored in this table.

To represent data flows, instances of the object **Method_Flow_Object** store information in the table **TBL_METHOD_FLOW**. Once again, this table contains mostly foreign key references.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_FROM_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_TO_ENTITY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_FLOW_INDEX</td>
<td>Integer</td>
</tr>
</tbody>
</table>
In order to associate data flows with entities, this table contains mostly foreign key values. Thus, the function of this table is to associate all data flows from one entity to another.

TBL_METHOD_FLOWTYPE is a lookup table for data flow types. References to the values in this table exist from the previously mentioned table.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FLOW_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FLOW_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_DESCRIPTION</td>
<td>String</td>
</tr>
</tbody>
</table>

Every data flow identified by the designer will have one entry in this table. The data flow name and a short description for that data flow may be stored in this table.

The methodology component in this study is assumed to store all new information at one point in time. This is analogous to the CASE tool component.

The trigger definition presented here to inform the feedback loop of an integration request is very much the same as the trigger definition presented for the CASE tool component:

```
Trigger TriggerMethodologyUpdate

   for <List_of_Methodology_Tables>
   on INSERT
   as
   INSERT
   <Control_Information>
   into TBL_FEEDBACK_CONTROL;
```
The trigger executes whenever information is inserted into the methodology component's database tables. This is indicated by `<List_of_Methodology_Tables>`. The action taken by the trigger is to create a request for integration in the control table TBL_FEEDBACK_CONTROL. The integration request is once again indicated by `<Control_Information>`.

8.5 Conclusion

The table definitions presented in this chapter allow information contained in the objects as defined in chapters 6 and 7 to be stored. The table layouts are not exhaustive. They are presented simply for the sake of completeness and to underline the importance of the repository for the feedback loop. The feedback loop is discussed in detail in the next chapter.
CHAPTER 9: THE FEEDBACK LOOP

9.1 Introduction

The environment for the feedback loop has been defined in the previous chapters. The components that play a major role were seen to be the CASE component, the methodology component and the repository.

The structure and extent of both the CASE and methodology components were discussed in detail. The ideas and definitions from those chapters are now employed to describe and explain the result of this study, the feedback loop.

9.2 An overview of the feedback loop

The term feedback loop has been chosen to indicate its function. Even though the concept of the feedback loop structure could be thought of as an abstract concept, this chapter will describe a practical implementation. The function performed by this structure is the bridging of the gap between the CASE tool and the methodology. The feedback loop allows the union between these two components to be created. In effect, the feedback loop creates the environment consisting of the CASE tool component and the methodology component, and integrates these components.

Even so, the feedback loop is much more than simply a bonding agent. It is a structure that allows the dynamic exchange of information between the CASE tool component and the methodology component within the environment. As was discussed earlier, the basis for an ideal scenario is the object-oriented paradigm.
As an example of the dynamic data exchange, consider the introduction of a new diagram into the system. The methodology prescribes that a certain diagram be created. Such a request is sent to the CASE tool component and from there a request is issued to the user. Once the developer has completed the diagram and committed it to the repository, the CASE tool informs the methodology component of this fact. This information exchange occurs via the feedback loop. The feedback loop has the responsibility of keeping both components at the same status.

9.3 Defining the functions of the feedback loop

Up to this point, the functions to be provided by the feedback loop have not yet been mentioned. The concept of integration should be mentioned before moving on to the major functions of the feedback loop. The reason for this should be clear: the feedback loop is an integration mechanism.

9.3.1 Integration demands

A number of different tools already exist to aid developers and to support software engineering. As [DALE93] points out, these tools all have the following in common:

1. they come from different vendors
2. they originate on different platforms and operating systems and are ported to other platforms and operating systems
3. they are based on different philosophies and employ different styles
4. they treat data as a proprietary tool
5. they have their own priorities and timetables for enhancements of their functionality
CHAPTER 9: THE FEEDBACK LOOP

A number of these statements are not useful for purposes of this study. Especially those mentioning vendors, operating systems and platforms are not applicable and not in the scope of this study.

The other points are interesting and should be investigated in more detail, especially with regard to the impact they may have on the feedback loop. The impact these have on the feedback loop's design and implementation will also be mentioned.

**a) Tools are based on different philosophies and employ different styles**

This is a fact that is common in all areas of the information technology industry. The reason for this is the distinction that each vendor requires of his product to be able to market and sell it. In other words, every vendor creates his own "look and feel" for his product set. This is natural, since this is an obvious way of differentiating products from other vendors.

Additionally, every vendor has his own method of completing tasks and integrates ideas that are used by him in his products. Just as a variety of software engineering philosophies exist, so a variety of tools exist to employ these philosophies and to make them available. With different philosophies come different styles.

As an example, one vendor streamlines development of software according to certain guidelines that include how much development can be afforded and what list of functions must be supported. [KEYE92]

The feedback loop is not concerned with the "look and feel" of a component in the environment. As long as the component conforms to certain standards with regard to information interchange, the feedback loop can be used to interface
with these components. The standards such a component conforms to must be defined. Not any data format is usable, for example. The feedback loop will be fully defined somewhat later in this chapter. It will be seen that the format for information interchange is directly dependent on the structure of tables in the repository. The repository structures will be defined and must be adhered to if any component is required to integrate with the tool set.

Furthermore, it is desirable that the component is object-oriented due to the way in which the feedback loop is constructed.

b) Tools treat data as a proprietary tool

This is possibly the most serious problem plaguing available tools. In order to fit in with a certain product set, data is often stored in such a form that it is available only to other tools in the same toolset. This restricts the usability of the tool and restricts the developer in the tools he can choose. Integration is made especially difficult if data is stored in proprietary formats. A number of export and import functions may be supplied with certain tools. If these are not available, developers may need to create such functions to transfer data among different tools.

The feedback loop relies solely on information interchange to be able to integrate components within the environment. If data is stored in various formats that are not mutually compatible, the feedback loop cannot be used and integration is not achieved among the various components. This ties into the comments made previously. As was stated, the repository structures employed for data storage are the data format definitions and must be adhered to.
Thus, the ideal scenario is a uniform data format and storage pattern. If this is the case, the feedback loop must simply be setup to "understand" the information. It should be clear that information is the driving force behind the feedback loop mechanism.

c) Tools have their own priorities and timetables for enhancements

The vendor providing and developing a tool sets the timetable for its development. This includes possible enhancements and additional functionality. The developer making use of the tool is at the mercy of the vendor when it comes to functions that are required for work on a project. Usually, only influential customers have any say in the long-term development cycle of a commercial product.

The feedback loop is not concerned with issues relating to vendors. Instead, information must be present in a recognisable form. Any enhancements that are made to components within the environment impact the environment. The reason for this is that new functionality may be added and old functionality removed. If the integration among the components was previously based on a certain structure being in place, removing or altering that structure changes or destroys the integration. Enhancements and modifications must be made in such a way that compatibility is still guaranteed. Thus, changes must be made known to the feedback loop to accommodate change within the environment.
9.3.2 The integration framework

Given that the points discussed above hold true, toolsets used for development need to be employed in a consistent and applicable fashion. Thus, the developer must be able to rely on the fact that the tools to be integrated via the feedback loop must behave by:

- presenting a uniform interface
- sharing data among all tools in the toolset
- conform to the view of the software engineering process
- being controlled in a uniform manner

[DALE93] states that all these requirements would need to be met by each tool in order for every tool to be used in conjunction with any other tool. This is not possible since each point stated is very complex. Additionally, different solutions are available for meeting each of the points stated above. According to [DALE93], an integration framework must be created to provide solutions for the toolset integration. The feedback loop proposed in this study attempts to create such an integration framework.

9.3.3 Concepts of integration frameworks

Tools selected for a certain project need to be integrated. Reasons for this have been stated previously. It was indicated that for tools to integrate with each other, an integration framework must be defined or created. Once the integration framework has been created, the tools are expected to fit into that framework. [DALE93] states that the selection of the most appropriate framework for the project's needs will be a determining factor in identification of the tools needed.
The process of selecting a framework for integration can be likened to the selection of hardware and software. Usually, software will be selected first. Next, the appropriate hardware must be selected to run the software. It is possible to select hardware first and then select from a range of software for that particular hardware. In a similar fashion, the framework may be created first, with the required tools being selected to fit that framework. However, the tools may be picked first and the framework last.

The model developed in this study requires any tool selected by the designer to be registered within the global repository. In other words, information about the tool must be stored in the repository and the tool must be able to interact with the repository. Information sharing via the repository is very important for the feedback loop to function.

[DALE93] discusses a number of different integration possibilities, ranging from presentation integration to process integration. The following diagram illustrates these possibilities.
The different integration services will be discussed shortly, as they form the basis for any CASE integration framework.

**a) Presentation integration**

Presentation integration refers to the interface with the user of the tool. The interface may be for the purpose of invoking the tool or communicating with the tool as it performs its job.

It is common for modern tools to conform to certain interface standards. As noted by [NIEL92], consistency, an important usability characteristic, should apply across all media forming the total user interface. Two examples are the popular OSF/Motif standard for UNIX systems and the Microsoft Windows standard for PC workstations. According to [SNOW92], the wide acceptance of GUI's has forced the development of software for these platforms. Conformance to standards means that the user does not need to relearn the functions of a different tool. Since most functions will be performed in similar ways, the user can switch among a number of tools all from different vendors with relative ease. The stricter the conformance to standards, the easier the presentation integration can be accomplished.

The feedback loop is not concerned with the method information is conveyed to the user. In other words, if the CASE tool has a graphical user interface the feedback loop will work no different than if the CASE tool had a character based interface. The feedback loop's operation is hidden from the user.
b) Data integration

Data is one of the most important aspects of any tool. In order for tools to work together, data sharing is the most critical aspect. As an example, one tool may export information in a certain format. In order for another tool to read the information, it must support the export format of the other tool. In cases where the structure of exported information changes, the other tools must be informed of the changes.

[PAGE93] notes that integration with other tools is vitally important. The tool should be integrated with other tools in such a way that no inconsistencies can be introduced into the overall system storehouse. [PAGE93] The seamlessness of the integration should also preserve consistency. No by-passes should exist for developers to access data other than accessing the design.

According to [DALE93], data integration often fails due to the following facts

- interfaces do not accommodate changes easily without causing other interfaces to be impacted. As one tool changes its output interface, so other tools must change. Co-ordinating these changes becomes especially difficult if when one tool produces output that is to be used as input to a number of other tools.
- the interface for exporting and importing information is usually very narrow; usually it is limited to the features that the developer thought to be important
- as the number of tools participating in the interchange of information increase, so the problem of keeping the interface stable also increase. The number of integrations required can grow so large that the integration becomes impossible
CHAPTER 9: THE FEEDBACK LOOP

c) Control integration

Combining tools together to form a closed environment requires mechanisms to control tools by offering mechanisms to register tools, invoke individual tools, perform tool composition and allow messages to be passed among different tools. In order for messages to be passed between tools, registration of objects is required, as well as registration of the format of information exported by one tool and the format of information imports for other tools.

Tool registration is mentioned as the most important part of having a CASE framework by [DALE93]. Every tool within the framework must be registered in detail to be usable within the framework. The ability to suspend the working of a tool, or the replacement of a tool with a newer or different tool is also important.

If true flexibility is expected in the CASE framework it is important that different tools can take over different functions according to the user's requirements. Triggers within repositories are an efficient mechanism to provide true flexibility. Access to some tool could cause an activation of another tool, for example. It is important to understand that the combination of triggers and message passing mechanisms within the repository are not contradictory. The action of the trigger may be the invocation of a message, for example. Triggers play an important part in the construction and operation of the feedback loop.

The definition of triggers was discussed in chapter 8. In that chapter, it was shown how a trigger could be initiated automatically from within the repository once defined. Trigger execution is linked to a certain action being taken on some database table, such as the insertion or deletion of a record. Once that event has occurred, the defined trigger is executed and performs some action. Trigger definitions specific to the feedback loop will be presented later in this chapter.
c) Control integration

Combining tools together to form a closed environment requires mechanisms to control tools by offering mechanisms to register tools, invoke individual tools, perform tool composition and allow messages to be passed among different tools. In order for messages to be passed between tools, registration of objects is required, as well as registration of the format of information exported by one tool and the format of information imports for other tools.

Tool registration is mentioned as the most important part of having a CASE framework by [DALE93]. Every tool within the framework must be registered in detail to be usable within the framework. The ability to suspend the working of a tool, or the replacement of a tool with a newer or different tool is also important.

If true flexibility is expected in the CASE framework it is important that different tools can take over different functions according to the user's requirements. Triggers within repositories are an efficient mechanism to provide true flexibility. Access to some tool could cause an activation of another tool, for example. It is important to understand that the combination of triggers and message passing mechanisms within the repository are not contradictory. The action of the trigger may be the invocation of a message, for example. Triggers play an important part in the construction and operation of the feedback loop.

The definition of triggers was discussed in chapter 8. In that chapter, it was shown how a trigger could be initiated automatically from within the repository once defined. Trigger execution is linked to a certain action being taken on some database table, such as the insertion or deletion of a record. Once that event has occurred, the defined trigger is executed and performs some action. Trigger definitions specific to the feedback loop will be presented later in this chapter.
d) Process integration

Data integration can be described as being the most complex integration issue. In contrast, process integration is the least mature. [DALE93] A process may be defined as a set of partially ordered steps intended to reach a goal. [CURT92]

The results of process integration should be the modelling of processes, managing work-flows and identifying deliverables. As stated by [CURT92], the modelling of information systems has traditionally focused on analysing data flows and transformations which accounted only for the organisation's data and that portion of its processes involved with the data. According to [CURT92], successfully integrating systems into an enterprise often requires modelling even the manual organisational processes into which these systems intervene.

The services of process integration will allow the process that objects within the repository must follow, to be modelled more closely. This is to ensure that they follow an optimal path for maximum productivity, quality and effectiveness. The concept of process integration is a task for the feedback loop.

e) Platform integration

The integration of software on different hardware and operating system platforms is a very important issue. Developers are often forced to create products that are capable of running on a wide range of platforms to satisfy customer requirements. The emphasis of software engineering has shifted from one to many platforms, including the mainframe, mini computer and personal computer platforms. [KEYE92]

Platform integration in its simplest form can be thought of as being the provision that tools and their interfaces do not change when moved from one platform to
another. In other words, the look and feel, and especially all aspects of its operation, of a product should remain unchanged. The topic of platform integration is not considered in this study.

f) Environment integration

The term environment integration refers to the combination of all the other integration services. A product set offering most of the above integration services can be said to offer environment integration. In order to add a new tool into the framework the changes required to that tool should be minimal. The tool should already provide and make use of most of the integration services.

9.4 The feedback loop as an integration framework

The definition of all actions and duties the feedback loop should be able to perform is vital to the further discussion of the feedback loop. These can be summed up as being:

1. to provide information exchange between the two components in the environment, namely the CASE tool and the software engineering methodology
2. to provide mechanisms for keeping the information available to both components in a consistent state and to provide mechanisms to store global information
3. to form a bond between the CASE tool and the software engineering methodology by monitoring the actions performed and to react to these actions

A discussion of these functions follows.
9.4.1 Information exchange

Information exchange between the two components of the environment is the primary and most important function the feedback loop must provide. The exchange of information forms the basis for the integration of the two separate components. Without information exchange, the two components are separate and no feedback from one component to the other is possible.

It is important to notice that the concept of information interchange ties directly into the concept of data integration as discussed above. To recap, the concept of data integration, or the data integration service, needs to ensure that information from one component of the toolset is usable by any other component within the toolset. A number of methods can be employed to achieve this, the most notable and important of which was the introduction of a data repository. The concept of a repository has been discussed in a previous chapter.

9.4.2 Maintain consistency and store global information

To become an integrated whole, both components must be fed information as work progresses. The continuous exchange of information introduces the danger of messages being lost or not received by the destination component. Once information is not received by the destination component, problems arise. A certain task may not have been marked as completed but should have been. Inconsistencies of this nature mean that the integration of the components has failed. The feedback loop has the responsibility to ensure that this cannot happen.

The feedback loop can be likened to an integration service. The previous function or service the feedback loop needs to offer was seen to be the data integration service.
The concept of maintaining consistency in all information available to all components is a further function the feedback loop must offer. A part of this service is the data integration service. This service must not only ensure that information is not lost as it is transferred from one component to the other, but also that the information is transferred remains intact and contains the original message. Thus, a controlling mechanism is required. The control integration service ensures that tool combinations are able to function in a consistent fashion. It was illustrated earlier that the registration of tools is very important. Additionally, triggers may be employed together with message based concepts to ensure the usability and viability of a toolset combination.

Storing the global information is accomplished rather simply by employing a data repository. [DALE93] has already stated that this is the most elegant solution. The repository provides the advantage of being a storage location for all global information that must be shared. The data integration service can function more efficiently when provided with a repository. Furthermore, the control integration service requires a database to store its triggers, message originators and receivers as well as registration information.
9.4.3 Integrating the CASE tool and the methodology

The previous point has touched on this subject to some degree. The feedback loop must integrate the two components of the environment. The CASE tool must have information as to what the methodology dictates to be the next phase or step. In the same way, the methodology must be informed as to which steps that have been completed. The requirement for this integration is purely the ability of both components to be able to communicate. Information exchange forms the basis for this function, and redefines two separate components to form an integrated whole.

If this function of the feedback loop is examined with the integration services listed above in mind, it is clear that the integration service required here is the concept of the environment integration service. The environment integration service is too complex and all-encompassing to be considered here. The feedback loop is not concerned with such concepts as presentation and platform integration services since these services are specific to hardware and operating systems. Thus, the feedback loop is unable to offer the range of services that environment integration entails.
9.5 Defining the feedback loop in terms of an integration framework

The previous paragraphs have refined the concept of the feedback loop, as well as its functions. By making use of an integration framework that provides integration services and applying these services to the concept of the feedback loop, a precise definition for the feedback loop can be obtained.

Thus, the feedback loop is defined as follows:

the feedback loop is an environment integration service between a CASE tool component and a software engineering methodology component in a software engineering environment that provides an integration framework with data integration services, control integration services and process integration services

The definition states that the feedback loop is a mechanism for integrating two components, namely a CASE tool component and a software engineering methodology component. The feedback loop forms the integration framework for combining these components by providing data integration services for information exchange and control integration services for ensuring that the components can be used together in a consistent fashion. Both components are used within a software engineering environment, which means that further tools could or could not be present. Whether or not further tools are present depends purely one whether or not these tools provide integration services in accordance to those of the feedback loop's integration framework.

Above, the features to be included in the feedback loop were discussed. The following section describes the technical aspects of creating the feedback loop in more detail.
methods in the object-oriented sense. The term service may be likened to the object-oriented term with the same name - here a service is a method that completes certain tasks as required. [PINS88]

9.7 Creating the feedback loop

Both components that are to interact with each other have been completely defined. In chapter 8 the most important components for the feedback loop's construction were discussed. It was shown how table structures and triggers could be used. The repository is an important building block for the feedback loop. Thus, the components taking part in the feedback loop's operation are

- the CASE component
- the methodology component
- the repository

Since the repository plays the biggest part in the integration via the feedback loop, the structures required will now be defined.

9.7.1 Repository structures for the feedback loop

The repository consists of a number of database tables and a data dictionary. The tables required for the CASE component and the methodology component have been defined in chapter 8. These tables are used primarily by the objects making up the CASE or methodology component in question. The feedback loop requires at least one structure within the repository. This is a table which is monitored by the feedback loop itself. The reason for this table's existence will now be explained.
The feedback loop is implemented as a daemon or background process. This can be likened to a daemon process in a UNIX environment or a memory resident program on a PC workstation; on a PC workstation such a process or program is often referred to as a TSR (Terminate and Stay Resident). In order for the feedback loop to be activated, one or more entries are placed in a special table in the repository. The feedback loop is active and checks that table for entries. Should it find an entry, the feedback loop carries out the instruction as read from the table. If no entry is found in the table, the feedback loop process remains dormant for a short period of time and then checks the table for entries again. Using the table as a control structure means that the feedback loop is completely independent of any actions that are to be taken. Furthermore, this means that the feedback loop can be used with CASE and methodology components that are freely selectable according to the criteria introduced earlier.

The control table used by the feedback loop will be called TBL_FEEDBACK_CONTROL. The definition of this table makes use of the conventions introduced earlier. This table is implemented as a relational table in the repository with the following structure:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ENTRY_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_ENTRY_TIME</td>
<td>Time</td>
</tr>
<tr>
<td>FLD_REQUESTOR</td>
<td>String</td>
</tr>
<tr>
<td>FLD_fk_SOURCE_COMPONENT</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_fk_DESTINATION_COMPONENT</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_fk_ACTION_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_IN_PROGRESS</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

The table layout is very simple. It contains information regarding the entry. The index for the entry in this table guarantees that each entry made in the table is unique. The index is a sequential number an is assigned automatically once an insert command is carried out. The fields pertaining to the date and time of the
entry, FLD_ENTRY_DATE and FLD_ENTRY_TIME, are used for administrative purposes. In a similar vein, the field FLD_REQUESTOR contains information regarding the person or workstation used to create or change something.

FLD_FK_SOURCE_COMPONENT and FL_FK_DESTINATION_COMPONENT are foreign keys that refer to a catalogue table. FLD_FK_ACTION_ID is also a reference to a catalogue table containing all possible actions that may be taken.

The field FLD_ACTION_IN_PROGRESS indicates whether or not the feedback loop has responded to the entry. If the contents of this field is TRUE, the feedback loop has already submitted a request to the destination component and the entry will be ignored. If the contents of the field is FALSE, the feedback loop has not yet noticed the entry in the table. In other words, the feedback loop has not reached the stage of its next execution.

The layout of the catalogue or reference table TBL_COMPONENTS is as follows:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_COMPONENT_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_COMPONENT_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_COMPONENT_VENDOR</td>
<td>String</td>
</tr>
<tr>
<td>FLD_COMPONENT_DESCRIPTION</td>
<td>String</td>
</tr>
<tr>
<td>FLD_COMPONENT_VERSION</td>
<td>String</td>
</tr>
<tr>
<td>FLD_COMPONENT_DATE_INIT</td>
<td>Date</td>
</tr>
</tbody>
</table>

Once again, the index field is used to uniquely identify each entry and to provide automatic primary key generation. FLD_COMPONENT_ID is an identification number assigned to each component for internal use. It is this field that ties back to the field FLD_FK_SOURCE_COMPONENT, for example. In this way the component can be identified immediately. Various information is stored for
CHAPTER 9: THE FEEDBACK LOOP

each component, such as the name, vendor, version and a description. The date the component was initialised for the feedback loop is stored in the field FLD_COMPONENT_DATE_INIT.

The layout of the other reference table TBL_ACTIONS is as follows:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_NAME</td>
<td>String</td>
</tr>
<tr>
<td>FLD_ACTION_DESCRIPTION</td>
<td>String</td>
</tr>
<tr>
<td>FLD_FK_ACTION_BY_COMPONENT_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_COMMAND</td>
<td>String</td>
</tr>
</tbody>
</table>

TBL_ACTIONS is another reference table, this time for the permitted actions that the feedback loop may activate. FLD_ENTRY_INDEX is the primary key field and is initialised automatically upon a new entry. The FLD_ACTION_ID field allows references to the field FLD_FK_ACTION_ID in the feedback loop table TBL_FEEDBACK_CONTROL, for example. Every action in the table has a name and a description. The most important fields in this table are the fields FLD_FK_ACTION_BY_COMPONENT_ID and FLD_ACTION_COMMAND. The first of these indicates whether or not the action requested may actually be performed by the feedback loop. Thus, an illegal action can be prevented from occurring. A lookup or reference table is used to determine legal and illegal actions. The action as given by the contents of FLD_ACTION_COMMAND is carried out by the feedback loop once the validity of the action has been determined.
CHAPTER 9: THE FEEDBACK LOOP

The last lookup table in use by the feedback loop is TBLVALID_ACTIONS.

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_SOURCE_COMPONENT</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_DESTINATION_COMPONENT</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_VALID</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

The fields FLD_ACTION_ID, FLD_FK_SOURCE_COMPONENT and FLD_FK_DESTINATION_COMPONENT are references to the other tables in use by the feedback loop. The field FLD_ACTION_VALID determines whether or not the action is permitted or not. To this end the source and destination components must be known. In other words, a certain action may be permitted from the CASE component to the methodology component but the same action may not be permitted from the methodology component to the CASE component, for example.

The tables described above will collectively be known as the feedback loop control tables to distinguish them from the other tables in the repository.

9.7.2 Using the feedback loop control tables

Before the feedback loop itself is further defined, its basic operation will be described. As was stated before, the control tables play a major part in the operation of the feedback loop. The following diagram illustrates how the feedback loop works. In the example shown in the illustration, a change is initiated from the CASE component. The change is propagated to the methodology component by the feedback loop.
An explanation of this diagram is required; the numbers in the diagram correspond with the following discussion:

1. the CASE tool component requests an update to its database tables. The reason for this could be a number of things, depending on the intention of the developer. Assume that the developer or designer has just completed the construction of a data flow diagram. Once the developer is satisfied that the diagram represents what he or she had in mind, the developer saves the diagram definitions.

2. the CASE tool component knows that all diagram definitions are to be stored in tables dedicated to diagram information. The object responsible for this writes the information to those tables.
CHAPTER 9: THE FEEDBACK LOOP

3. the tables for the CASE tool component have database triggers associated with them. This was explained in the chapter dealing with the repository. Different triggers are associated with the various tables. Depending on the changes made by the developer, one or more triggers may be activated to transfer information to the methodology component.

4. the triggers associated with the CASE component tables update the feedback loop control tables with appropriate entries.

5. the feedback loop checks its control tables for something to do. Finding the entries that were placed in the table by the triggers from the CASE component tables, the feedback loop validates the entries.

6. if all the entries in the control table are valid and permissible, the feedback loop dispatches a message to the appropriate objects in the methodology component. The entries in the control table are marked to indicate that they are being attended to.

7. the objects in the methodology component receive the request and start their operation according to the action they were supplied with by the feedback loop. Assuming that no user input is required, each action is performed in the sequence it was supplied and with any additional information supplied.

8. once the actions have been successfully completed, the objects in the methodology component update the methodology component tables. Additionally, the feedback control tables receive a status that indicates the status of the update request. This is done with the aid of database triggers. If the update request was successful, the control table entries are removed. Should an error have occurred, the entries in the control table could be marked as waiting for dispatch; this could make provision for retries of the same request should some component currently be busy.

9. it is important to realise that the update of tables for one component can cause another update request for another component. It is also possible that a number of requests may never be noticed by the user. The feedback loop is thus capable of maintaining the environment automatically.
CHAPTER 9: THE FEEDBACK LOOP

The example presented above has illustrated how the feedback loop operates. Next, the dependency among the various components is discussed.

9.7.3 Interdependency within one component

The example presented above has dealt only with the information transfer from the CASE tool component to the methodology component. A number of interdependencies exist within each of these components, however. As an example, the developer or designer must follow the phases of the methodology as prescribed by the methodology. In other words, the designer must carry out all actions as prescribed by a certain phase in the methodology before moving on to the next phase.

As an example of this, consider the methodology component as defined in chapter 7. The first phase of the methodology component prescribes that all entities within the problem space are to be identified. Assume the developer skips this phase. The next phase states that all attributes are to be identified for all entities. This is impossible if the developer has skipped the preceding phase. In order to avoid this, the methodology component presents the phases in the order the work is to be carried out. Once the developer has completed the tasks for a particular phase, he or she indicates that the current phase has been completed. The methodology component then allows the developer to continue with the next phase.

This brings a further problem to the fore. Assume that the developer realises halfway through a certain phase that a number of additional entities are required, or that some entities are not required. Thus, it must be possible for the developer to backtrack and change definitions already made.
In order for this to be possible, the component itself must be responsible. The most important concept that must be understood concerning the feedback loop's operation is that it is responsible for the integration of the CASE and methodology components. It is not the task of the feedback loop to ensure integrity within the components of the toolsets. If this were the case, the feedback loop itself would be a methodology component or a CASE component. The feedback loop ensures integrity of the information shared by the CASE and methodology components.

9.7.4 Revisiting diagram modelling transparency

The concept of integrating diagrams by means of modelling transparency was introduced in chapter 5 as an important aspect for the model developed in this study. Diagram integration is important for the feedback loop because of information exchange possibilities. Information sharing by means of a repository was shown to be a very important aspect of diagram integration.

The sequential dependency aspect discussed in chapter 5 is important for the CASE and methodology components interacting by way of the feedback loop. Recall from the discussion of sequential dependencies that a sequential dependency exists if some component must be brought into agreement with another component. This is exactly what the feedback loop is aiming to achieve. Once a change is made to an object in the CASE tool component, the feedback loop is responsible for bringing the objects within the methodology component into agreement with the changes made.

In order to implement these concepts, the object-oriented approach has been implemented together with a global repository for information storage.
9.8 Defining the feedback loop class hierarchy

The discussion concerning the feedback loop should have clearly defined its operation requirements. In the following sections, a complete object class definition for the feedback loop will be presented. The conventions used for the naming of objects as well as the Get and Set methods introduced earlier are used. Since the feedback loop is the focus of this study, a more detailed description will be given of its internal operation. In most cases, very basic pseudocode will be used to define the feedback loop's operation. There is no need for the reader to know programming syntax to understand these definitions - all code samples will be fully explained and examples will be presented.

The conventions used to create the method definitions are the following. The body of the code segment is bound by the name of the method and the keywords Begin and End. A semicolon is used to indicate the end of a statement.

```
Sample_Method
    Begin;

    End;
```
CHAPTER 9: THE FEEDBACK LOOP

Any variables or temporary storage requirements are defined in the same way as for the object definitions. An integer required within the method to hold some value is defined and used as in the following example:

```
Sample_Method
    SampleInt_Var of type Integer;
    Begin;
        SampleInt_Var := 0;
    End;
```

All variables are temporary and exist only within the confines of the method they have been defined in.

Object-oriented specific conventions should also be mentioned. Whenever a call to a method is made from within a method, the method call is made up of the object instance name followed by a period followed by the method name and possibly a parameter list. As an example:

```
Sample_Method
    SampleInt_Var of type Integer;
    Begin;
        SampleInt_Var := 0;
        Other_Object.Other_Method;
    End;
```

Whenever a method is called that has been defined for the current object, the object instance name may be replaced by the word Self. Self is simply a reference to the current object instance. Another object that is assumed to exist within the environment is the object instance System. The System object
provides basic functions and services such as returning the current date, time, process identification and others.

General programming constructs and embedded SQL statements are used. All statements are explained.

The parent object definition, which is the uppermost in the feedback loop object hierarchy, is defined as follows:

\[ Object \text{ Feedback\_Parent} \]

It is important to note that the feedback loop is primarily table-driven. In other words, the feedback loop mechanism is idle whenever there are no new entries in the control tables. The feedback loop engine requires some default information regarding its execution interval.

\[ Object \text{ FeedbackBase\_Object(Feedback\_Parent)} \]
\[ IdleTime\_Attribute \text{ of type Integer} \]
\[ SetIdleTime\_Method \]
\[ GetIdleTime\_Method \]
\[ CheckInstance\_Method \]

An instance of this object "knows" at what intervals the control tables are to be checked. The attribute \text{IdleTime\_Attribute} is set in number of seconds. Thus, given a value of 20, the feedback loop will check its control tables every 20 seconds.

It is very important to understand that only one instance of the feedback loop may be running at one time. The reason for this should be clear - the feedback loop attempts to communicate information from the methodology component to
the CASE component and vice versa. If a second feedback loop instance exists, the operation of the control tables cannot be guaranteed. The feedback loop marks those actions in the control tables that are in operation. Only one process is capable of ensuring that the actions are applied in the correct sequence. It is for this purpose that the method \textit{CheckInstance\_Method} exists.

The method \textit{CheckInstance\_Method} has the task of ensuring that only one feedback loop instance is active. If another instance is found to be active, the current feedback loop instance is terminated or killed.

\textbf{CheckInstance\_Method} \\
\texttt{EntryCount\_Var of type Integer;} \\
\texttt{Begin;} \\
\texttt{ EntryCount\_Var = SELECT COUNT(*) FROM TBL\_ACTIVE\_INSTANCE;} \\
\texttt{ If EntryCount\_Var > 1 then} \\
\texttt{ Begin;} \\
\texttt{ Self.Destruct;} \\
\texttt{ End;} \\
\texttt{ End;} \\

The table TBL\_ACTIVE\_INSTANCE belongs to the feedback loop control tables and has the following structure within the repository:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_INSTANCE_PID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_START_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_START_TIME</td>
<td>Time</td>
</tr>
</tbody>
</table>
CHAPTER 9: THE FEEDBACK LOOP

The table contains an entry for the feedback loop detailing the time and date the process was started and the process identification number (PID) it has been assigned.

The method CheckInstance_Method requires a local variable to hold the result of an embedded query statement. The number of entries within the table described above is retrieved from the repository. If the number of entries is found to be greater than 1, it is taken as an indication that a feedback loop instance is already running. In this case the current instance is terminated with the statement Self.Destruct. The use of Self has been explained to refer to the current object instance. The keyword Destruct is generic to the object-oriented programming paradigm. It is an instruction which is associated with each object definition to allow cleanup of resources, such as memory, for that object instance. Thus, the statement Self.Destruct removes the entire instance of Self from memory and thereby terminates the instance.

As an aside, the generic method Construct is used to create an actual instance from an object definition.
The actual feedback loop instance is created from the definition for the Feedback Object.

Object Feedback_Object(FeedbackBase_Object)

  CheckPointTime_Attribute of type Time
  Init_Method
  SetCheckPointTime_Method
  GetCheckPointTime_Method
  CheckIdleTime_Method
  CheckControlTable_Method
  PerformAction_Method
  IsActionValid_Method
  DoAction_Method
  HandleEvent_Method

Even though the structure for this object definition seems simple, the work performed by the feedback loop is contained within the methods for this object. In particular, it will be seen that the method HandleEvent_Method is the method that keeps the feedback loop in operation. A detailed description of this object and all method definitions should present a clear picture of the operation of the feedback loop.
CHAPTER 9: THE FEEDBACK LOOP

Init_Method

PID_Var of type Integer;
StartTime_Var of type Time;
StartDate_Var of type Date;
Begin;

FeedbackBase_Object.CheckInstance_Method;
PID_Var := System.GetPID();
StartTime_Var := System.GetTime();
StartDate_Var := System.GetDate();
INSERT INTO TBL_ACTIVE_INSTANCE
(FLD_INSTANCE_PID, FLD_START_DATE,
 FLD_START_TIME)
VALUES
(:PID_Var, :StartTime_Var, :StartDate_Var);
Self.SetCheckPointTime_Method;
End;

Methods with the name Init have become a standard feature in object-oriented programs. By convention, these methods contain the startup code to initialise the object instance.

The first step in the feedback loop instance's initialisation is a check for an already existing instance. The CheckInstance_Method has already been described. If the current instance is found to be the only instance in existence, the Init_Method continues. The services or methods GetTime, GetDate and GetPID are called. As already explained, the System object provides system related services to all the other objects. The values returned by the system are stored in the feedback loop control table TBL_ACTIVE_INSTANCE by means of an embedded SQL statement for future reference and especially for the method
CHAPTER 9: THE FEEDBACK LOOP

CheckInstance_Method. As a final initialisation requirement, the method SetCheckPointTime_Method is called.

```
SetCheckPointTime_Method
   CurrentTime_Var of type Time;
   Begin;
      CurrentTime_Var := System.GetTime();
      Self.CheckPointTime_Attribute := CurrentTime_Var;
   End;
```

The feedback loop executes only once its idle time has passed. In order for the idle time to be usable, a checkpoint time is stored within the object. Thus, the checkpoint time is used in conjunction with the idle time to determine whether the feedback loop should execute or not.

The method SetCheckPointTime_Method retrieves the current time as set in the system. The time is stored in the attribute that is accessible to all methods within the class hierarchy, the CheckPointTime_Attribute.

```
GetCheckPointTime_Method
   Begin;
      Return Self.CheckPointTime;
   End;
```

The method GetCheckPointTime is called with a parameter. The keyword Return ensures that the value to be returned by the method is placed in the parameter provided by the calling object. As can be seen, this method simply returns the value of the checkpoint time to the method that is requesting the information.
CheckIdleTime_Method

CurrentTime_Var of type Time;
CheckPointTime_Var of type Time;
IdleTime_Var of type Integer;
Begin;

Self.GetCheckPointTime_Method CheckPointTime_Var;
IdleTime_Var := FeedbackBase_Object.IdleTime_Attribute;
CurrentTime_Var := System.GetTime();
If (CheckPointTime_Var + IdleTime_Var) >=
CurrentTime_Var then
Begin;

Return TRUE;
Self.SetCheckPointTime_Method;

Else;

Return FALSE;
End;

End;

CheckIdleTime_Method determines whether or not the feedback loop is to be awakened or not. In order to do this, both the checkpoint time and the idle time are retrieved into local storage. The local variables CurrentTime_Var and CheckPointTime_Var contain the respective values. In order to determine the current time, the system service GetTime is consulted. The value is placed into a local variable CurrentTime_Var.

Recall that the feedback loop executes whenever its idle time has elapsed. Thus, a calculation is performed to determine whether or not this is the case. The calculation (CheckPointTime_Var + IdleTime_Var) returns a value that is compared to the current time. If the value calculated is greater or equal to the current time, it can be stated that the idle time has elapsed. In this case, the method returns the Boolean value TRUE and resets the checkpoint time. If the
idle time has not yet elapsed, the method simply returns the Boolean value FALSE. It will be illustrated shortly how these Boolean values are used.

```
CheckControlTable_Method
EntryCount_Var of type Integer;
Begin;
    EntryCount_Var = SELECT COUNT(*) FROM TBL_FEEDBACK_CONTROL WHERE FLD_ACTION_IN_PROGRESS = FALSE;
    If EntryCount_Var > 0 then
        Begin;
            Self.PerformAction_Method;
        End;
End;
```

The method CheckControlTable_Method determines whether there are any entries in the primary feedback loop control table TBL_FEEDBACK_CONTROL. Recall that the feedback loop marks entries that are being attended to. This is done by placing the entry TRUE in the field FLD_ACTION_IN_PROGRESS. If an action has not yet been attended to by the feedback loop, the field contains the Boolean value FALSE. Thus, the CheckControlTable_Method determines the number of entries in the control table that the feedback loop has not yet responded to.
If one or more entries exist that must be attended to, the method `PerformAction_Method` is called to perform the required action.

```
PerformAction_Method
ActionCursor_Var of type Cursor;
ActionID_Var of type Integer;
ActionDestination_Var of type Integer;
ActionValid_Var of type Boolean;
Begin;
    ActionCursor_Var := CREATE CURSOR FOR UPDATE AS
    SELECT * FROM TBL_FEEDBACK_CONTROL
    WHERE FLD_ACTION_IN_PROGRESS = FALSE
    ORDER BY FLD_ENTRY_INDEX;
    ActionCursor_Var.Open;
    ActionCursor_Var.MoveFirst;
    Repeat;
    ActionCursor.FLD_ACTION_IN_PROGRESS := TRUE;
    ActionID_Var := ActionCursor.FLD_FK_ACTION_ID;
    ActionDestination_Var := ActionCursor.
    FLD_FK_DESTINATION_COMPONENT;
    Self.IsActionValid_Method ActionID_Var,
    ActionValid_Var;
    If ActionValid_Var = TRUE then
    Begin;
    Self.DoAction_Method ActionID ActionID_Var
    ActionDestination_Var;
```
Else;
    System.Write "Error in action" + ActionID_Var;
    ActionCursor.FLD_ACTION_IN_PROGRESS := FALSE;
End;
ActionCursor_Var.MoveNext;
Until ActionCursor_Var.Status = END;
ActionCursor_Var.Close;
End;

The definition for the method PerformAction_Method is the most complex. It relies on two further methods that are discussed below. This method represents the feedback loop's functionality. In other words, to inform the other component in the environment, either the CASE tool component or the methodology component, of a change.

The operation of the method is fairly straightforward. Since there may be more than one operation that the feedback loop must process, a cursor is opened. A cursor can be likened to a virtual table or a temporary result set. In other words, the cursor is a special structure that contains one or more resulting rows of a query. The special commands or keywords to manipulate the cursor rows are:

- **Open** - to create the result set or cursor
- **MoveFirst** - to move the very first row contained in the cursor
- **MoveNext** - to move from the current row to the next row in the cursor
- **Close** - to destroy the cursor and to deallocate memory resources allocated to it

Once the temporary variable has been created as a cursor, the fields in the rows of the cursor are accessible simply by naming them. It is important to note that the cursor allows updates to the table to be made.
Thus, the method creates a cursor that contains all actions the feedback loop must respond to and the first row in the result is selected by means of the command *MoveFirst*. A loop is executed in which each row in the cursor is examined in turn. The same processing steps are applied to each row in the cursor. As each row is accessed, the field FLD_ACTION_IN_PROGRESS is set to prevent any inconsistencies.

The *ActionID* and *ActionDestination* contain the identification of the action to be performed and the component that is to perform the action, respectively. In order to determine whether the action that is required is permissible, the method *IsActionValid_Method* is called with the *ActionID* as a parameter. This method returns a Boolean value that indicates whether the feedback loop may process the action.

If the result returned is TRUE, the *DoAction_Method* is called upon to perform the required action. If the result returned is FALSE, an error has occurred. The error message is logged and the entry for FLD_ACTION_IN_PROGRESS is reset. An error could occur due to a time-out in the destination component or the database. In the worst case, an error has occurred because a component has violated system integrity. All actions specified by any component must be permissible - possibly the component has been updated with new or changed functionality without being registered with the feedback loop control tables.

Once all rows in the cursor have been consulted, the cursor is closed and the feedback loop is idle once more.
To determine whether an action to be taken by the feedback loop is valid, the following method is employed:

\[ \text{IsActionValid\_Method } \text{ActionID\_Par} \]
\[ \text{EntryCount\_Var of type Integer;} \]
\[ \text{Begin;} \]
\[ \text{EntryCount\_Var := SELECT COUNT(*) FROM} \]
\[ \text{TBL\_VALID\_ACTIONS WHERE FLD\_ACTION\_ID =} \]
\[ \text{ActionID\_Par;} \]
\[ \text{If EntryCount\_Var > 0 then} \]
\[ \text{Begin;} \]
\[ \text{Return TRUE} \]
\[ \text{Else;} \]
\[ \text{Return FALSE;} \]
\[ \text{End;} \]
\[ \text{End;} \]

The method \text{IsValidAction\_Method} determines whether a certain action may be performed or not. By selecting the number of entries that contain the action identification number, it is possible to determine the validity of an action. If the action is valid, the value TRUE is returned, otherwise the value FALSE is returned. As has been mentioned, all actions should be permissible. A system integrity violation is the most probable cause and should be investigated as soon as possible.
CHAPTER 9: THE FEEDBACK LOOP

DoAction_Method ActionID_Par, ActionDestination_Par
ActionCommand_Var of type String;
Begin;
    ActionCommand_Var := SELECT
        FLD_ACTION_COMMAND FROM TBL_ACTIONS
        WHERE FLD_ACTION_ID = ActionID_Par;
    ActionDestination_Par.ActionCommand;
End;

The DoAction_Method is the method that ultimately performs the work. Once all checks have been performed without returning errors, this method calls the appropriate object in the destination component together with the message read from the control table. The destination object thus performs the work of updating itself and its surrounding objects.

HandleEvent_Method
Decide_Var of type Boolean;
Begin;
    Repeat
        CheckIdleTime_Method Decide_Var;
        If Decide_Var = TRUE then
            Begin;
                CheckControlTable_Method;
                End;
        Until FALSE;
    End;

The handle event method is an accepted object-oriented programming concept. It is this method that is responsible for maintaining the object's execution and checking for certain conditions. The HandleEvent_Method for the feedback loop
continuously checks the status of the idle time. Once the idle time has elapsed, the feedback loop is called into operation. Note that the loop within the event handler is continuous. Execution of the object is stopped only once the system is shut down.

9.9 Communicating change to the feedback loop

The preceding discussion has defined the feedback loop in detail. The feedback loop monitors certain control tables and takes an action depending on the state of the information within those tables. This section will briefly focus on the information transfer from either the CASE or the methodology component to the feedback loop. Once this discussion has been presented, the entire environment will be able to function without intervention. This discussion includes an example.

The feedback loop executes in cycles as defined by the setting of its IdleTime_Attribute. If no entries exist in the control table TBL_FEEDBACK_CONTROL, the feedback loop returns to its idle state. If entries exist, all those entries are examined and processed.

A number of triggers must exist to fill this feedback loop control table. In particular, the tables related to both the CASE and the methodology component must have triggers associated with them. This section will present a short overview of the triggers required. Defining and identifying all required triggers is beyond the scope of this study. The tables in use by the CASE and methodology components were presented in chapter 8. Please refer back to those definitions if required.
CHAPTER 9: THE FEEDBACK LOOP

Assume the following scenario. A designer starts the design process by making use of an automated methodology tool MethodStep. The methodology proposes the following steps:

1. define entities in the problem space
2. define attributes for all entities identified
3. define relationships among all entities identified

The designer investigates the problem space and determines all important entities. The designer informs the methodology that the first phase has been completed. It is at this point that information must be communicated to the CASE tool. In this example, the CASE tool has the name CASEStep. Entry of the identified entities is necessary.

The object Method_MethodPhase_Object calls its method Set_PhaseComplete_Method to indicate that this particular phase has been completed. In other words, the value of the field FLD_PHASE_COMPLETE is set to TRUE in the table TBL_METHOD_METHODPHASE. In order to find the most recent action that has been completed, the field FLD_TIME_COMPLETE is initialised with the system time at the time of the task's completion.
The following trigger must exist in the repository to inform the CASE tool of the status change:

```
Trigger_TriggerCaseEntityEntry
for TBL_METHOD_METHODPHASE
on UPDATE TO FLD_PHASE_COMPLETE
as
(SELECT FLD_PHASE_ID
    INTO :PHASE_ID
    FROM TBL_METHOD_METHODPHASE
    WHERE FLD_TIME_COMPLETE =
        MAX(FLD_TIME_COMPLETE);
SELECT FLD_COMPONENT_ID
    INTO :SOURCE FROM TBL_COMPONENTS
    WHERE FLD_COMPONENT_NAME = 'METHODSTEP';
SELECT FLD_COMPONENT_ID
    INTO :DESTINATION FROM TBL_COMPONENTS
    WHERE FLD_COMPONENT_NAME = 'CASESTEP';
INSERT (SYSTEM.DATE, SYSTEM.TIME,
    SYSTEM.WORKSTATION, :SOURCE, :DESTINATION,
    :PHASE_ID, FALSE)
    INTO TBL_FEEDBACK_CONTROL);
```

The trigger definition for `Trigger_TriggerCaseEntityEntry` is necessarily complex and has been simplified somewhat for purposes of this discussion.

The trigger is executed whenever an update occurs to one of the rows in the table `TBL_METHOD_METHODPHASE`. In particular, the trigger is executed only if the field `TABLE_PHASE_COMPLETE` is changed. The only change of this
field is from a value of FALSE to one of TRUE. This occurs whenever a phase has been completed.

As a first step, the trigger retrieves the value of the fields TBL_PHASE_ID from the most recently updated row. The selection statement relies on the maximum value function to find this row; in other words, the row with the latest time in the field FLD_TIME_COMPLETE was the most recently updated row. Note that the value of the phase identification resulting from the query statement is placed in a temporary variable for use later on.

In the following two steps, queries retrieve the identification numbers for both the source and the destination component involved in this operation. The source component in this example is the methodology component and the destination component is the CASE tool. These values are also placed into temporary variables.

The final step is to insert a new entry into the feedback loop control table. Values for the date, time and workstation identification are retrieved from the current system settings. The activity identifier FLD_FK_ACTION_ID is assumed to be the same as the phase identifier in this example. The value FALSE is entered for the field FLD_ACTION_IN_PROGRESS to ensure that the feedback loop reads and responds to the entry once it restarts.
Once the trigger has executed, the feedback control table TBL_FEEDBACK_CONTROL contains at least the following entries, for example:

<table>
<thead>
<tr>
<th>FLD_ENTRY_INDEX</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_DATE</td>
<td>12/4/95</td>
</tr>
<tr>
<td>FLD_ENTRY_TIME</td>
<td>12:53</td>
</tr>
<tr>
<td>FLD_REQUESTOR</td>
<td>WKST_1</td>
</tr>
<tr>
<td>FLD_FK_SOURCE_COMPONENT</td>
<td>1</td>
</tr>
<tr>
<td>FLD_FK_DESTINATION_COMPONENT</td>
<td>2</td>
</tr>
<tr>
<td>FLD_FK_ACTION_ID</td>
<td>1</td>
</tr>
<tr>
<td>FLD_ACTION_IN_PROGRESS</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

The example assumes that this is the third entry for the control table. The necessary entries for the other tables in use by the feedback loop will be presented next.

The reference table TBL_COMPONENTS contains registration information for the methodology and the CASE components. As an example, the entry for the methodology component MethodStep could be:

<table>
<thead>
<tr>
<th>FLD_ENTRY_INDEX</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_COMPONENT_ID</td>
<td>1</td>
</tr>
<tr>
<td>FLD_COMPONENT_NAME</td>
<td>METHODSTEP</td>
</tr>
<tr>
<td>FLD_COMPONENT_VENDOR</td>
<td>SOMEWARE INC.</td>
</tr>
<tr>
<td>FLD_COMPONENT_DESCRIPTION</td>
<td>AUTOMATED METHODOLOGY TOOL</td>
</tr>
<tr>
<td>FLD_COMPONENT_VERSION</td>
<td>2.2.0</td>
</tr>
<tr>
<td>FLD_COMPONENT_DATE_INIT</td>
<td>1/3/95</td>
</tr>
</tbody>
</table>
CHAPTER 9: THE FEEDBACK LOOP

As was noted earlier, all actions the feedback loop may perform are stored in the table TBL_ACTIONS. An entry for this table pertaining to the example above could be:

<table>
<thead>
<tr>
<th>FLD_ENTRY_INDEX</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ACTION_ID</td>
<td>1</td>
</tr>
<tr>
<td>FLD_ACTION_NAME</td>
<td>CASE_ENTITY_REQUEST</td>
</tr>
<tr>
<td>FLD_ACTION_DESCRIPTION</td>
<td>ENTITY DEFINITION REQUEST</td>
</tr>
<tr>
<td>FLD_FK_ACTION_BY_COMPONENT_ID</td>
<td>1</td>
</tr>
<tr>
<td>FLD_ACTION_COMMAND</td>
<td>CALL CASE_Object.InputEntities</td>
</tr>
</tbody>
</table>

It is assumed that the CASE component has some method, such as InputEntities, that makes it possible to initialise a subcomponent for entry of entity definitions. Should this not be the case, multiple calls to methods such as the SetEntityName_Method could be executed. This is not ideal - the CASE tool should provide a module that allows the user to enter entity information. By calling that module directly all other necessary actions in the CASE tool are also involved. This ensures the integrity of the information processed by the CASE tool.

An important table for validating actions to be taken by the feedback loop is the table TBL_VALID_ACTIONS. For the transfer of information to the CASE tool in the example above to be possible, an entry similar to the following example should exist:

<table>
<thead>
<tr>
<th>FLD_ENTRY_INDEX</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ACTION_ID</td>
<td>1</td>
</tr>
<tr>
<td>FLD_FK_SOURCE_COMPONENT</td>
<td>1</td>
</tr>
<tr>
<td>FLD_FK_DESTINATION_COMPONENT</td>
<td>2</td>
</tr>
<tr>
<td>FLD_ACTION_VALID</td>
<td>TRUE</td>
</tr>
</tbody>
</table>
Given that the table entries exist as above, the feedback loop will execute and cause the required event. This event is a prompt to the user in the CASE tool to enter the required entity definitions. If the designer were to start entering attribute definitions for these entities as well, for example, the feedback loop would communicate this fact to the methodology component. In this way, both components are updated in real time without requiring operator intervention.

9.10 Conclusion

This chapter has presented a detailed discussion of the feedback loop and its possible implementation. The feedback loop was constructed to operate with two components that were defined during the course of chapters 6 and 7. In those chapters, the CASE tool component and the methodology component were constructed according to the object-oriented paradigm. In chapter 8, the repository was introduced as a third necessary component. The required database structures were defined for the CASE tool component and the methodology component. The structures were mainly tables that would be used to store information.

In this chapter, all the components and structures introduced in those earlier chapters were formed into one coherent unit. With the aid of examples the operation of the feedback loop was explained.
CHAPTER 10: CONCLUSION

10.1 Introduction

The aim of this study was to define some method of integration between a CASE tool and an automated software engineering methodology tool. The structure that resulted to solve this problem was the so-called feedback loop. The feedback loop was based primarily on the object-oriented paradigm and employed database tables and triggers.

The feedback loop was introduced in a step-by-step fashion - all the underlying and important concepts and structures were introduced in successive chapters.

10.2 Summarising the study

This study has attempted to construct a feasible base on which future studies can build. The aim of this study was not to create a fully-featured software development environment, but rather to illustrate the shortcomings of current technologies and to suggest an alternative.

The feedback loop developed during the course of this study has been constructed from existing components. The object-oriented paradigm has been proven to be more useful than the older procedural paradigm. The object-oriented paradigm was found to contain all the necessary features to build the feedback loop as it was presented here.

The relational database management systems currently in use contain useful components such as triggers and stored procedures. These were found to be necessary and useful for the construction of the repository. In particular, the
CHAPTER 10: CONCLUSION

Trigger mechanisms allow the repository to function unaided. In other words, the repository is not dependant on programs having to be written and maintaining the repository from the outside. This ensures transparency to the CASE and methodology components interacting via the repository.

The concept of the object database management system was mentioned but not fully explored. The ODMS has numerous advantages above the relational and extended relational database management systems. One big advantage is the integration of the object-oriented paradigm directly into the database. Given that the object database management system is capable of supporting all the features currently supported by an RDBMS, as well as all the language features of an OOPL, the feedback loop mechanism could be implemented with tighter integration among the components of the environment.

The study has attempted to create an ideal environment in which designers and developers can be more productive and keep a tighter reign on the project they are currently busy with. Unfortunately, in doing so the model developed here had to make certain assumptions. One such assumption is that any currently available CASE tool can be placed on a new repository structure. Even though a number of the newer CASE tools are repository independent, the structures in use by these CASE tools are created by the vendor of the tool. This means that the feedback loop control tables may have to be modified to be able to interact with that tool. The same argument may be applied to automated methodology tools. It may furthermore prove difficult to find both a CASE tool and an automated methodology that are capable of running from the same repository, bearing in mind that both could have been purchased from separate vendors.

Making use of the chosen tools in a cross-platform or heterogeneous environment may also be difficult. An example of this is a MS-DOS-based software methodology tool and a UNIX-based CASE tool. Another example is a
proprietary flat file format in use by the methodology tool and a relational
database in use by the CASE tool. The feedback loop is not capable of dealing
with these issues in its present form.

The feedback loop may nonetheless be viewed as an example of making
existing technology work for the end-user. In particular, the feedback loop is a
tool that can aid the project team be on time and on budget - with the security of
knowing that information entered into one component has not been lost or
forgotten.

As with any theoretical model, the feedback loop mechanism can be proven to
be effective only once it has been used in the "real-world".

10.3 Future areas of investigation

The focus of this study has been the theoretical concept of integrating a CASE
tool and a methodology component. All the concepts introduced in this study for
the construction of the feedback loop are based on existing areas of technology.
Thus, the next area of investigation would be the actual implementation of the
feedback loop using the object definitions presented in chapters 6,7 and 9
together with the table definitions presented in chapters 8 and 9 as a foundation.

A practical implementation of the feedback loop should show benefits for most
CASE - methodology interaction.
BIBLIOGRAPHY

[ADCY91] AD/Cycle Digest, Second Edition
Fall 1991
International Business Machines Corporation

[AVIS91] D.E. Avison
MERISE in Practice
Macmillan Publishing Ltd.
1991

[BARK90] R. Barker
CASE*Method - Tasks and Deliverables
1990
Addison-Wesley

[BELL92] D. Bell, J. Grimson
Distributed database systems
Addison-Wesley
1992

[BERT93] E. Bertino, L. Martino
Object-oriented database systems
Addison-Wesley
1993

[BOOC86] G. Booch
Object-oriented Development
IEEE Transactions on Software Engineering
Vol. SE-12 No. 2
1986

[BRIN93] S. Brinkkemper
Integrating diagrams in CASE Tools through Modelling
Transparency
Information and software technology
Vol. 35 No. 2
February, 1993

[CASE86] A.F. Case
Information Systems Development: Principles of Computer-Aided
Software Engineering
Prentice-Hall, 1986
<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publisher</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>[DATE90]</td>
<td>C.J. Date</td>
<td>An introduction to database systems Volume 1</td>
<td>Addison-Wesley</td>
<td>1990</td>
</tr>
</tbody>
</table>
BIBLIOGRAPHY

[FISH] Fisher
Technological Trends in CASE
John Wiley & Sons

[GEHA86] N. Gehani, A.D. McGettrick
Software Specification Techniques
Addison-Wesley Publishing Company
1986

[GILB88] T. Gilb
Principles of Software Engineering Management
Addison-Wesley Publishing Company
1988

[HARE94] J.S. Hares, J.D. Smart
Object orientation - Technology, Techniques, Management and Migration
John Wiley & Sons
1994

[HARM91] P. Harmon, B. Sawyer
ObjectCraft - A graphical programming tool for object-oriented applications
Addison-Wesley
1991

[HOAR75] C.A.R. Hoare
The Engineering of Software: A Startling Contradiction
British Computer Society
1975

[HUMP89] W.S. Humphrey
CASE Planning and the Software Process
Managing the Software Process
Addison-Wesley
1989

[IEFA90] IEF - A Totally Integrated CASE Environment
1990
Texas Instruments Incorporated

[IEFD91] IEF Development Coordination Overview
Version 1.0
Texas Instruments Incorporated
[IEFT90] IEF Technology Overview
Texas Instruments Incorporated

[INCE89] D.C. Ince
Software engineering
Van Nostrand Reinhold
1989

[JOHN89] J.R. Johnson
The Software Factory - Managing Software Development and Maintenance
QED Information Sciences
1989

[JONE80] C.B. Jones
Software Development - A Rigorous Approach
1980
Prentice Hall International

[KEYE92] J. Keyes
New metrics needed for a new generation
Software Magazine
May 1992

[LANO94] K. Lano and H. Haughton
Object-oriented Specification Case Studies
Prentice Hall
1994

[LEDG87] H. Ledgard, J. Tauer
Software Engineering Concepts
Addison-Wesley
1987

[LOCH89] F. Lochovsky, W. Kim
Object-oriented Concepts, Databases, and Applications
ACM Press
Addison-Wesley Publishing Company
1989

[MACR87] A. Macro, J. Buxton
The Craft of Software Engineering
Addison-Wesley
1987
[MAIR93] P. Mair
The Strategic Procurement of CASE Tools
NCC Blackwell
1993

[MAND92] D. Mandrioli, B. Meyer
Advances in Object-oriented Software Engineering
Prentice Hall
1992

Microsoft
1993

[MITC90] R.J. Mitchell
Managing Complexity in Software Engineering
Peter Peregrinus
1990

[MOIG77] J.L. Le Moigne
La theorie du systeme generale
PUF
Paris
1977

[MULL90] M. Mullin
Rapid prototyping for object-oriented systems
Addison-Wesley
1990

The usability engineering life cycle
Computer, Vol. 25, No. 3
March 1992

[OLLE88] T.W. Olle et al
Information Systems Methodologies
Addison-Wesley
1988

[ORAC] R. Stewart et al
Oracle CASE - From Decision to Use
Oracle Corporation
[OULD90] M.A. Ould
Strategies for Software Engineering - The Management of Risk and Quality
1990
John Wiley & Sons Ltd.

[OXFO83] J.B. Sykes
The Concise Oxford Dictionary of Current English
7th Edition
University Press, 1983

[PAGE93] M. Page-Jones
The CASE Manifesto
Computer Language
June 1993

[PINS88] L.J. Pinson, R.S. Wiener
An Introduction to Object-oriented Programming and Smalltalk
Addison-Wesley Publishing Company
1988

[PRAT87] P.J. Pratt, J.J. Adamski
Database Systems - Management and Design
1987
Boyd & Fraser Publishing Company

[PRES87] R.S. Pressman
Software Engineering: A Practioner's Approach
Mc-Graw Hill Book Company
1987

[QUAN91] P.T. Quang, C. Chartier-Kastler
MERISE in Practice
Macmillan Publishing Ltd.
1991

[ROSN75] J. de Rosnay
Le macroscope
Editions du Seuil
Paris
1975
[SAGA90] J.M. Sagawa  
Repository Manager Technology  
IBM Systems Journal  
Volume 29, Number 2  
1990

[SALC93] L.M. Salchenberger  
Structured development techniques for user-developed systems  
Information & Management, No. 24  
1993

[SAVI90] D. Savic  
Object-oriented Programming with Smalltalk/V  
Ellis Horwood Publishing  
1990

[SHUM90] A.F. Shumskas  
Prerequisites for Successful Software Development  
Experience with the Management of Software Projects 1989  
Edited by F. J. Mowle and P. F. Elzer  
International Federation of Automatic Control Workshop Series  
Number 9, 1990

[SNEE89] H.M. Sneed  
Software Engineering Management  
Ellis Horwood Limited  
1989

[SNOW92] D. Snow  
Windows test tools catching up  
Software Magazine  
May 1992

[SODH91] J. Sodhi  
Software Engineering Methods, Management, and CASE Tools  
TAB Books  
1991

[SOMM92] I. Sommerville  
Software Engineering  
Addison-Wesley Publishing Ltd.
[SØLV93] A. Sølvberg, D.C. Kung
Information systems modelling - An introduction
Springer-Verlag
1993

[STEV91] W. Stevens
Software Design - Concepts and Methods
Prentice Hall
1991

[TELL89] E.R. Tello
Object-oriented Programming for Artificial Intelligence
Addison Wesley
1989

[TKAC94] D. Tkach, R. Puttick
Object Technology in Application Development
IBM International Technical Support Organisation
The Benjamin/Cummins Publishing Company
1994

[TURS87] W.M. Turski, T.S.E. Maibaum
The Specification of Computer Programs
Addison-Wesley Publishing Company
1987

[WARD92] A. Ward
The Next Generation of Computer Assistance for Software Engineering
ACM SIGSOFT
Software Engineering Notes Vol. 17 No. 3
July 1992

Object-oriented Software
Addison-Wesley Publishing Company
1990
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE</td>
<td>Computer aided software engineering</td>
</tr>
<tr>
<td>CFD</td>
<td>Control flow diagram</td>
</tr>
<tr>
<td>DBMS</td>
<td>Database management system</td>
</tr>
<tr>
<td>DFD</td>
<td>Data flow diagram</td>
</tr>
<tr>
<td>ERD</td>
<td>Entity-relationship diagram</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical user interface</td>
</tr>
<tr>
<td>I-CASE</td>
<td>Integrated computer aided software engineering</td>
</tr>
<tr>
<td>ICT</td>
<td>Intelligent CASE tool</td>
</tr>
<tr>
<td>ODMS</td>
<td>Object data management system</td>
</tr>
<tr>
<td>OOD</td>
<td>Object-oriented design</td>
</tr>
<tr>
<td>OOP</td>
<td>Object-oriented programming</td>
</tr>
<tr>
<td>OOPL</td>
<td>Object-oriented programming language</td>
</tr>
<tr>
<td>OSF</td>
<td>Open Systems Foundation</td>
</tr>
<tr>
<td>PC</td>
<td>Personal computer</td>
</tr>
<tr>
<td>RAD</td>
<td>Rapid application development</td>
</tr>
<tr>
<td>RDBMS</td>
<td>Relational database management system</td>
</tr>
<tr>
<td>SC</td>
<td>Structured chart</td>
</tr>
<tr>
<td>SQL</td>
<td>Structured query language</td>
</tr>
<tr>
<td>TSR</td>
<td>Terminate stay resident</td>
</tr>
</tbody>
</table>
INDEX

A
attribute, 57, 58, 59, 61, 62, 65, 78, 138, 139, 145, 178, 183, 197
automated, 10, 12, 15, 18, 19, 40, 43, 48, 78, 192, 198, 199

C
CASE, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 37, 39, 40, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 54, 55, 56, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 74, 75, 76, 78, 79, 80, 81, 82, 83, 84, 86, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 101, 102, 103, 104, 120, 127, 136, 137, 138, 139, 140, 141, 142, 145, 147, 149, 150, 156, 159, 161, 164, 165, 167, 168, 171, 172, 173, 174, 175, 179, 187, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200
class, 52, 53, 55, 62, 63, 64, 65, 78, 86, 87, 88, 100, 113, 114, 117, 118, 119, 120, 121, 122, 123, 125, 126, 127, 132, 135, 176, 183
communication, 4, 8, 29, 40, 42, 49, 52, 54, 61, 62, 63
component, 6, 18, 26, 40, 42, 45, 48, 49, 51, 52, 54, 55, 56, 61, 62, 63, 64, 65, 66, 68, 69, 71, 74, 75, 76, 78, 79, 81, 82, 83, 86, 101, 102, 103, 104, 113, 118, 120, 125, 126, 127, 149, 150, 151, 152, 162, 163, 165, 167, 169, 171, 172, 173, 174, 175, 178, 187, 188, 190, 191, 194, 195, 196, 197, 200
dependency, 13, 57, 74, 75, 76, 77, 78, 174, 175
design, 1, 3, 4, 5, 8, 10, 12, 14, 15, 17, 18, 19, 20, 22, 23, 24, 25, 26, 28, 30, 32, 35, 36, 37, 38, 39, 40, 43, 45, 46, 47, 48, 49, 50, 51, 52, 68, 69, 70, 71, 74, 84, 85, 103, 112, 151, 157, 192
development, 1, 3, 5, 8, 9, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 29, 30, 31, 32, 33, 35, 37, 38, 43, 44, 45, 46, 47, 48, 49, 51, 68, 70, 71, 87, 103, 104, 131, 151, 153, 154, 156, 198
diagram, 25, 33, 47, 50, 53, 55, 57, 61, 64, 66, 67, 68, 69, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 84, 85, 86, 87, 88, 89, 90, 92, 95, 96, 99, 100, 109, 111, 112, 120, 126,
dbms, 130, 135
data flow, 47, 67, 70, 71, 73, 75, 80, 84, 85, 86, 89, 91, 95, 96, 97, 98, 104, 111, 112, 113, 120, 125, 126, 141, 147, 172

Page 210
INDEX

127, 137, 141, 150, 155, 166, 171, 172, 175
dictionary, 15, 25, 158, 167
dynamic, 2, 68, 149, 150

E
encapsulated, 62
engineering, 1, 2, 3, 4, 5, 6, 8, 9, 10, 19, 23, 25, 28, 42, 43, 44, 46, 49, 50, 69, 73, 102, 103, 104, 150, 151, 154, 160, 161, 165, 198
entity, 13, 52, 53, 57, 61, 67, 70, 71, 73, 76, 77, 80, 84, 85, 86, 89, 90, 91, 92, 93, 95, 97, 98, 104, 105, 106, 109, 111, 120, 121, 122, 123, 125, 131, 132, 138, 139, 140, 145, 146, 147, 196, 197
entity-relationship, 57, 84, 109
environment, 5, 6, 9, 10, 12, 14, 16, 17, 18, 19, 20, 21, 23, 26, 27, 32, 35, 42, 45, 47, 48, 49, 51, 54, 55, 61, 63, 65, 69, 70, 71, 75, 76, 78, 79, 81, 82, 83, 102, 104, 149, 151, 152, 153, 159, 161, 162, 164, 165, 166, 168, 173, 177, 187, 191, 198, 199
exchange, 2, 61, 68, 71, 74, 149, 150, 161, 162, 164, 165, 175

F
framework, 3, 19, 25, 154, 155, 156, 159, 161, 165

G
GUI, 2, 16, 17, 49

H
hierarchy, 53, 80, 81, 87, 88, 100, 126, 127, 132, 176, 178, 183
independence, 11, 18
inhereritance, 4, 49, 51, 52, 54, 64, 65, 66, 67, 76, 77, 80, 82, 95, 99, 100, 127, 135
integrated, 5, 9, 11, 40, 44, 48, 54, 69, 154, 157, 158, 162, 164
integration, 3, 5, 9, 11, 17, 21, 25, 26, 27, 35, 42, 43, 47, 48, 49, 50, 61, 68, 69, 80, 82, 101, 103, 128, 142, 147, 148, 150, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 165, 166, 167, 175, 198, 199

L
life cycle, 9, 12, 15, 21, 23, 27, 28, 29, 30, 31, 32, 33, 34, 36, 38, 40, 41

M
management, 4, 12, 13, 15, 17, 21, 27, 28, 31, 37, 43, 47, 49, 129, 130, 131, 132, 135, 198, 199
messages, 52, 53, 54, 133, 159, 162
INDEX

method, 14, 23, 51, 52, 57, 58, 59, 60, 61, 62, 63, 64, 65, 67, 71, 74, 78, 87, 97, 114, 141, 151, 156, 167, 176, 177, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 192, 196, 198


modelling, 3, 21, 49, 50, 51, 52, 53, 54, 55, 57, 68, 69, 71, 72, 73, 74, 76, 78, 79, 80, 81, 83, 84, 130, 160, 175

modularity, 39, 101, 127

O

object-oriented, 1, 39, 49, 50, 51, 52, 53, 54, 55, 56, 60, 62, 63, 64, 67, 68, 81, 82, 83, 85, 100, 101, 102, 105, 106, 127, 128, 130, 131, 135, 149, 152, 166, 167, 175, 180, 182, 190, 197, 198, 199

ODMS, 130, 131, 132, 133, 135, 136, 199

P

procedure, 31, 52, 60, 129, 132, 133

prototyping, 1, 11, 13, 14, 22, 69

R

RDBMS, 15, 199

reference, 15, 45, 68, 115, 129, 131, 137, 144, 145, 169, 170, 177, 182, 195

relational, 15, 129, 130, 131, 132, 135, 168, 198, 199, 200

repository, 6, 13, 15, 16, 23, 25, 68, 69, 71, 73, 74, 76, 78, 79, 80, 81, 83, 128, 129, 134, 135, 137, 141, 148, 149, 150, 152, 155, 158, 159, 160, 162, 163, 167, 168, 171, 173, 175, 179, 180, 193, 197, 198, 199

S

services, 23, 53, 54, 57, 59, 60, 67, 102, 156, 160, 161, 162, 164, 165, 166, 178, 182

software, 1, 2, 3, 4, 5, 6, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 23, 24, 25, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 55, 56, 62, 73, 80, 84, 87, 102, 103, 104, 105, 106, 150, 151, 154, 155, 156, 160, 161, 165, 198, 199

subclass, 53, 89, 92, 95

T

transparency, 49, 68, 69, 72, 73, 76, 78, 79, 81, 175, 199

trigger, 129, 132, 133, 134, 141, 142, 147, 148, 159, 193, 194, 195, 199

Page 212
INTEGRATION OF A CASE TOOL AND A SOFTWARE ENGINEERING METHODOLOGY

M.P. Hertenberger
Prof. J.H.P Eloff

Department Computer Science
Rand Afrikaans University
P.O. Box 524
Auckland Park, Johannesburg
South Africa

Abstract

Computer aided software engineering (CASE) tools alone are unable to support the software development life cycle. Such tools provide the means to create the final software product. In contrast, software engineering methodologies formalise the steps taken in the development cycle of a software product. However, the result of following the steps proposed by a software engineering methodology is not a final software product. Instead, the methodology allows the designer to produce a complete and detailed specification for the resulting software product. It is from this specification that the software product is eventually constructed from.

A number of studies have been carried out in both the fields of CASE tools and software engineering methodologies. However, very few studies have attempted to combine the two technologies in such a way that could enhance the productivity of designers and developers. It is the aim of this paper to briefly discuss the development of the feedback loop, a mechanism that achieves the integration of a CASE tool and a software engineering methodology.
1. Introduction

The aim of this paper is to provide a possible solution to the integration problem existing between a CASE tool and a software engineering methodology.

The problem that exists may be stated simply as follows: a number of software engineering methodologies exist. Also, a number of CASE tools exist. The methodology used by a developer is usually not specific to the CASE tool being used; conversely, the CASE tool does not take any particular methodology into account. The most important statement to be made from these facts is that the issue is not one of not having the tools, but rather how to use the tools in a consistent and integrated fashion. [DALE93] Stated simply, the problem definition states that software engineering methodologies and CASE tools stand apart.

Thus, the proposed model developed in this paper will attempt to tackle the integration of a software engineering methodology with a tool for Computer Aided Software Engineering. It is the aim of this study to create an environment in which each of the two components, the CASE tool and the methodology, are able to exert the same influence on the developer. Both should be used in all aspects of the design and development process, and they should be co-
operative in helping the developer reach the goal of creating a quality software product.

The software tools used for the creation of new software are still in a process of evolution. [MACR87] Similarly, software engineering methods are still being developed. [MACR87] However, object-orientation is the technology that is replacing today's database and programming technology for the design and development of computerised application systems - the technology that some regard as the ultimate paradigm for the modelling of information. [HARE94] The object-oriented paradigm will be used extensively in this paper.

2. A problem of integration

Just as a large number of CASE tools and CASE environments are available, so a large number of software engineering methodologies abound. Of the tools available, a number of tools cater only for specific requirements. Other tools offer a wealth of features. A number of possibilities and products exist for developers to create so-called toolsets - environments in which all tools have the ability to communicate with each other as the development process is carried out. [DALE93] However, the integration of these tools is still regarded as being in its infancy. [PAGE93]

Integration demands made on tools depend on the integration demands made by the developer. In certain situations, complete integration may be desired. In other situations, no integration may be desired. To this end, integration frameworks for tools must, and do, exist. [DALE93] Integration according to the chosen framework is always ideal. [DALE93] The goal pursued in this paper may be stated as finding a solution to the problem of integrating a CASE tool and a software engineering methodology.
APPENDIX

[DALE93] discusses a number of different integration possibilities, ranging from presentation integration to process integration. The following diagram illustrates these possibilities.

These integration possibilities will be described shortly:

- presentation integration refers to the interface with the user of the tool. The interface may be for the purpose of invoking the tool or communicating with the tool as it performs its job. Presentation integration is not considered in this paper.
- data is one of the most important aspects of any tool. In order for tools to work together, data sharing is the most critical aspect. This is referred to as data integration.
- control integration allows tools to be combined to form a closed environment. This requires mechanisms to control tools by offering mechanisms to register tools, invoke individual tools, perform tool composition and allow messages to be passed among different tools. In order for messages to be passed between tools, registration of objects is required, as well as registration of the format of information.
exported by one tool and the format of information imports for other tools.

- the results of process integration should be the modelling of processes, managing work-flows and identifying deliverables. As stated by [CURT92], the modelling of information systems has traditionally focused on analysing data flows and transformations which accounted only for the organisation's data and that portion of its processes involved with the data. According to [CURT92], successfully integrating systems into an enterprise often requires modelling even the manual organisational processes into which these systems intervene.

- platform integration in its simplest form can be thought of as being the provision that tools and their interfaces do not change when moved from one platform to another. In other words, the look and feel, and especially all aspects of its operation, of a product should remain unchanged. [KEYE92] The topic of platform integration is not considered in this paper.

The concepts of integration mentioned above will be applied to the feedback loop developed during the course of this paper.
3. The CASE tool component

The components to be integrated in this paper are some arbitrary CASE tool and an automated software engineering methodology. The CASE tool component is discussed briefly.

The definition that is given by [CASE86], is the following:

Computer-aided software engineering (CASE) is the application of automated technologies to the software engineering procedures.

It is this definition that is used in the remainder of this paper. The definition ensures that the entire range of software, from individual task-oriented tools to fully integrated operational software environments, is covered.

This paper is to focus on the construction of a model that incorporates a CASE tool component. Thus, it is important to include aspects of "real-world" CASE tools that could impact the feedback loop's operation. Of these the most important aspects are the concepts of verifiability, maintainability and project division and sub tasking.
4. The methodology component

Traditionally, software engineering methodologies were never available in a computerised form. Developers made use of lists and books containing the steps of the methodology they had chosen to use for a particular development task.

As the concept of using a methodology became more and more important, computerised versions of many popular methodologies became available. As an example, a number of implementations of the methodology MERISE exist, each one being slightly different where model design and vocabulary are used [QUAN91].

In order for the integration between the methodology component and the CASE tool component to be usable, the methodology component will cater for the following:

1. provision of tasks to request the identification of entities, attributes, relationships and data flows among entities
2. provision of tasks to request the creation of diagrams to include the entities, attributes, relationships and data flows identified by the user

The feedback loop discussed in this paper addresses these issues.
5. The feedback loop as an integration mechanism

According to [PAGE93], integration among tools is vitally important. The tools to be integrated in this paper have been identified as being the CASE tool and the methodology tool. The feedback loop is a mechanism that is to integrate these components.

Two of the integration methods that are relevant to the feedback loop are data integration and control integration.

5.1 Data integration

Data is one of the most important aspects of any tool. In order for tools to work together, data sharing is the most critical aspect. As an example, one tool may export information in a certain format. In order for another tool to read the information, it must support the export format of the other tool. In cases where the structure of exported information changes, the other tools must be informed of the changes.

The feedback loop mechanism developed in this paper makes use of repository structures to carry out its functions. For this approach to be feasible, the following must be taken into account:

- each tool providing information into the repository must be aware of the structure of the information within the repository. In order to accomplish this, a meta-model or description of the information within the data dictionary is required
- if the information within the repository is stored in the form of objects, the objects themselves together with information concerning the objects must be stored within the repository. The internals of the
objects, such as the attributes and methods provided by the object, are uninterpretable to tools not integrated with the repository.

- the definition of the repository in terms of objects must be considered a static definition, with information about how each tool manipulates the objects in the repository critical to understanding the dynamics of the repository. To this end, each tool must provide some explanation of how it manipulates objects within the repository.

- over time the versions of tools change and the definitions of objects within the repository also change. It is critical that the objects from older versions are updated or reformatted if any tool changes.

5.2 Control integration

Combining tools together to form a closed environment requires mechanisms to control tools by offering mechanisms to register tools, invoke individual tools, perform tool composition and allow messages to be passed among different tools. In order for messages to be passed between tools, registration of objects is required, as well as registration of the format of information exported by one tool and the format of information imports for other tools.

Tool registration is mentioned as the most important part of having a CASE framework by [DALE93]. Every tool within the framework must be registered in detail to be usable within the framework. The ability to suspend the working of a tool, or the replacement of a tool with a newer or different tool is also important. If true flexibility is expected in the CASE framework it is important that different tools can take over different functions according to the user's requirements. Triggers within repositories are an efficient mechanism to provide such flexibility.
6. Structures to create the feedback loop

The actual feedback loop mechanism may be defined as an active background process, or daemon, that is table driven. In other words, the feedback loop daemon is activated whenever entries are placed in special control tables in the repository. The entries in the control tables instruct the feedback loop to perform some action - this action integrates the CASE tool component and the methodology component.

The control table used by the feedback loop will be called TBL_FEEDBACK_CONTROL and is implemented as a relational table in the repository with the following structure:

<table>
<thead>
<tr>
<th>Field name</th>
<th>Data type</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLD_ENTRY_INDEX</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ENTRY_DATE</td>
<td>Date</td>
</tr>
<tr>
<td>FLD_ENTRY_TIME</td>
<td>Time</td>
</tr>
<tr>
<td>FLD_REQUESTOR</td>
<td>String</td>
</tr>
<tr>
<td>FLD_FK_SOURCE_COMPONENT</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_FK_DESTINATION_COMPONENT</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_ID</td>
<td>Integer</td>
</tr>
<tr>
<td>FLD_ACTION_IN_PROGRESS</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Each entry has a unique primary key guaranteed by the entry for FLD_ENTRY_INDEX. The date and time, as well as the person requesting the change are stored.
FLD_FK_SOURCE_COMPONENT and FL_FK_DESTINATION_COMPONENT are foreign keys that refer to a catalogue table. The entries in these fields determine the flow of information. FLD_FK_ACTION_ID is also a reference to a catalogue table containing all possible actions that may be taken. Actions are checked for validity in another lookup table. This prevents unauthorised actions to be carried out.

The field FLD_ACTION_IN_PROGRESS indicates whether or not the feedback loop has responded to the entry. If the contents of this field is TRUE, the feedback loop has already submitted a request to the destination component and the entry will be ignored. If the contents of the field is FALSE, the feedback loop has not yet noticed the entry in the table. In other words, the feedback loop has not reached the stage of its next execution.

The feedback loop is idle and executes at predetermined intervals. During its execution, the feedback loop checks the control table. If one or more actions are found to be pending, the requests are processed in sequence.

The structure of the feedback loop object definition is complex. Only the most relevant methods are presented here to give an overview of the feedback loop's operation.
Of these methods, the \texttt{CheckControlTable\_Method} is possibly the most important:

\begin{verbatim}
CheckControlTable\_Method  
EntryCount\_Var of type Integer;  
Begin;  
  EntryCount\_Var = SELECT COUNT(*) FROM  
    TBL\_FEEDBACK\_CONTROL WHERE  
    FLD\_ACTION\_IN\_PROGRESS = FALSE;  
  If EntryCount\_Var > 0 then  
    Begin;  
      Self\.PerformAction\_Method;  
    End;  
End;
\end{verbatim}

The method \texttt{CheckControlTable\_Method} determines whether there are any entries in the primary feedback loop control table \texttt{TBL\_FEEDBACK\_CONTROL}. Recall that the feedback loop marks entries that are being attended to. This is done by placing the entry TRUE in the field \texttt{FLD\_ACTION\_IN\_PROGRESS}. If an action has not yet been attended to by the feedback loop, the field contains the Boolean value FALSE. Thus, the \texttt{CheckControlTable\_Method} determines the number of entries in the control table that the feedback loop has not yet responded to. If one or more entries exist that must be attended to, the method \texttt{PerformAction\_Method} is called to perform the required action.

\begin{verbatim}
PerformAction\_Method  
  ActionCursor\_Var of type Cursor;  
  ActionID\_Var of type Integer;  
  ActionDestination\_Var of type Integer;  
  ActionValid\_Var of type Boolean;
\end{verbatim}
Begin;

    ActionCursor_Var := CREATE CURSOR FOR UPDATE AS
                        SELECT * FROM TBL_FEEDBACK_CONTROL
                        WHERE FLD_ACTION_IN_PROGRESS = FALSE
                        ORDER BY FLD_ENTRY_INDEX;
    ActionCursor_Var.Open;
    ActionCursor_Var.MoveFirst;
    Repeat;
        ActionCursor.FLD_ACTION_IN_PROGRESS := TRUE;
        ActionID_Var := ActionCursor.FLD_FK_ACTION_ID;
        ActionDestination_Var := ActionCursor.
                                FLD_FK_DESTINATION_COMPONENT;
        Self.IsActionValid_Method ActionID_Var,
            ActionValid_Var;
        If ActionValid_Var = TRUE then
            Begin;
                Self.DoAction_Method ActionID_Var,
                                 ActionDestination_Var;
            End;
        Else;
            System.Write "Error in action" + ActionID_Var;
            ActionCursor.FLD_ACTION_IN_PROGRESS := FALSE;
        End;
    ActionCursor_Var.MoveNext;
    Until ActionCursor_Var.Status = END;
    ActionCursor_Var.Close;

End;
The definition for the method PerformAction_Method is the most complex. It relies on two further methods that are discussed below. This method represents the feedback loop’s functionality. In other words, to inform the other component in the environment, either the CASE tool component or the methodology component, of a change.

The operation of the method is fairly straightforward. Since there may be more than one operation that the feedback loop must process, a cursor is opened. A cursor can be likened to a virtual table or a temporary result set. In other words, the cursor is a special structure that contains one or more resulting rows of a query. The special commands or keywords to manipulate the cursor rows are:

- **Open** - to create the result set or cursor
- **MoveFirst** - to move the very first row contained in the cursor
- **MoveNext** - to move from the current row to the next row in the cursor
- **Close** - to destroy the cursor and to deallocate memory resources allocated to it

Once the temporary variable has been created as a cursor, the fields in the rows of the cursor are accessible simply by naming them. It is important to note that the cursor allows updates to the table to be made.

Thus, the method creates a cursor that contains all actions the feedback loop must respond to and the first row in the result is selected by means of the command **MoveFirst**. A loop is executed in which each row in the cursor is examined in turn. The same processing steps are applied to each row in the cursor. As each row is accessed, the field FLD_ACTION_IN_PROGRESS is set to prevent any inconsistencies.

The ActionID and ActionDestination contain the identification of the action to be performed and the component that is to perform the action, respectively.
order to determine whether the action that is required is permissible, the method `IsActionValid_Method` is called with the `ActionID` as a parameter. This method returns a Boolean value that indicates whether the feedback loop may process the action.

If the result returned is TRUE, the `DoAction_Method` is called upon to perform the required action. If the result returned is FALSE, an error has occurred. The error message is logged and the entry for FLD_ACTION_IN_PROGRESS is reset. An error could occur due to a time-out in the destination component or the database. In the worst case, an error has occurred because a component has violated system integrity. All actions specified by any component must be permissible - possibly the component has been updated with new or changed functionality without being registered with the feedback loop control tables.

Once all rows in the cursor have been consulted, the cursor is closed and the feedback loop is idle once more.

The method `IsActionValid_Method` determines whether the action to be carried out by the feedback loop is valid. If the action is listed in the table `TBL_VALID_ACTIONS`, it may be executed.

The `DoAction_Method` is the method that ultimately performs the work. Once all checks have been performed without returning errors, this method calls the appropriate object in the destination component together with the message read from the control table. The destination object thus performs the work of updating itself and its surrounding objects.
DoAction_Method ActionID_Par, ActionDestination_Par
ActionCommand_Var of type String;
Begin;
    ActionCommand_Var := SELECT
                        FLD_ACTION_COMMAND FROM TBL_ACTIONS
                        WHERE FLD_ACTION_ID = ActionID_Par;
                        ActionDestination_Par.ActionCommand;
End;

The discussion of the feedback loop's structure has been shortened considerably and has focused on the most important aspects only. The following section presents a more detailed example of how the feedback loop could be used in a real world situation.

7. Operation of the feedback loop - an example

The most important structures used to create the feedback loop have been defined above. The following diagram illustrates how the feedback loop works. In the example shown in the illustration, a change is initiated from the CASE component. The change is propagated to the methodology component by the feedback loop.
An explanation of this diagram is required; the numbers in the diagram correspond with the following discussion:

1. the CASE tool component requests an update to its database tables. The reason for this could be a number of things, depending on the intention of the developer. Assume that the developer or designer has just completed the construction of a data flow diagram. Once the developer is satisfied that the diagram represents what he or she had in mind, the developer saves the diagram definitions.

2. the CASE tool component knows that all diagram definitions are to be stored in tables dedicated to diagram information. The object responsible for this
writes the information to those tables. Thus, the data flow diagram information is written to the relevant tables in the database.

3. the tables for the CASE tool component have database triggers associated with them. Different triggers are associated with the various tables. Depending on the changes made by the developer, one or more triggers may be activated to transfer information to the methodology component. If the data flow diagram has been modified, only the changes are communicated to the methodology component.

4. the triggers associated with the CASE component tables update the feedback loop control tables with appropriate entries.

5. the feedback loop checks its control tables for something to do. Finding the entries that were placed in the table by the triggers from the CASE component tables, the feedback loop validates the entries.

6. if all the entries in the control table are valid and permissible, the feedback loop dispatches a message to the appropriate objects in the methodology component. The entries in the control table are marked to indicate that they are being attended to.

7. the objects in the methodology component receive the request and start their operation according to the action they were supplied with by the feedback loop. Assuming that no user input is required, each action is performed in the sequence it was supplied and with any additional information supplied.

8. once the actions have been successfully completed, the objects in the methodology component update the methodology component tables. Additionally, the feedback control tables receive a status that indicates the status of the update request. This is done with the aid of database triggers. If the update request was successful, the control table entries are removed. Should an error have occurred, the entries in the control table could be marked as waiting for dispatch; this could make provision for retries of the same request should some component currently be busy.
9. It is important to realise that the update of tables for one component can cause another update request for another component. It is also possible that a number of requests may never be noticed by the user. The feedback loop is thus capable of maintaining the environment automatically.

The example presented above has been simplified. It does show how the feedback loop can be used to integrate the CASE tool and the methodology more tightly, however. The benefits of such an integration should be clear from the example.

8. Conclusion

The aim of this paper was to define some method of integration between a CASE tool and an automated software engineering methodology tool. The structure that resulted to solve this problem was the so-called feedback loop. The feedback loop was based primarily on the object-oriented paradigm and employed database tables and triggers.

The integration requirements as presented by [DALE93] were discussed. In particular, the concepts of data and control integration were shown to be relevant to the feedback loop. Given that the feedback loop is capable of solving the problems of data and control integration, the feedback loop is capable of integrating two or more components reliably. The components to be integrated in this paper were an arbitrary CASE tool and automated software engineering methodology.

The feedback loop developed in this paper has been constructed from existing components. The object-oriented paradigm has been proven to be more useful
than the older procedural paradigm as it was found to contain all the necessary features to build the feedback loop.

9. Bibliography

[CASE86] A.F. Case
Information Systems Development: Principles of Computer-Aided Software Engineering
Prentice-Hall, 1986

[COAD90] P. Coad, E. Yourdon
Object-oriented Analysis
Yourdon Press
Prentice Hall
1990

Process modelling
Communications of the ACM, Vol. 35, No. 9
September 1992

[DALE93] R. Daley
Integration Technology for CASE
UNICOM Applied Information Technology Series
1993

[GILB88] T. Gilb
Principles of Software Engineering Management
Addison-Wesley Publishing Company
1988

[HARE94] J.S. Hares, J.D. Smart
Object orientation - Technology, Techniques, Management and Migration
John Wiley & Sons
1994

[KEYE92] J. Keyes
New metrics needed for a new generation
Software Magazine
May 1992
APPENDIX

[MACR87] A. Macro, J. Buxton
The Craft of Software Engineering
Addison-Wesley
1987

[OLLE88] T.W. Olle et al
Information Systems Methodologies
Addison-Wesley
1988

[PAGE93] M. Page-Jones
The CASE Manifesto
Computer Language
June 1993

[QUAN91] P.T. Quang, C. Chartier-Kastler
MERISE in Practice
Macmillan Publishing Ltd.
1991

[SNEE89] H.M. Sneed
Software Engineering Management
Ellis Horwood Limited
1989

[TURS87] W.M. Turski, T.S.E. Maibaum
The Specification of Computer Programs
Addison-Wesley Publishing Company
1987