Knowledge-based Support for Object-oriented Design

by

MARIANNE LOOCK

submitted in partial fulfilment of

the requirements for the degree of

MASTER OF SCIENCE

in the subject

INFORMATION SYSTEMS

at the

UNIVERSITY OF SOUTH AFRICA

SUPERVISOR: PROFESSOR A.L. STEENKAMP

JUNE 1994

Abstract

The research is conducted in the area of Software Engineering, with emphasis on the design phase of the Software Development Life Cycle (SDLC). The object-oriented paradigm is the point of departure. The investigation deals with the problem of creating support for the design phase of object-oriented system development. This support must be able to guide the system designer through the design process, according to a sound design method, highlight opportunities for prototyping and point out where to re-iterate a design step, for example. A solution is proposed in the form of a knowledge-based support system. In the prototype this support guides a designer partially through the first step of the System Design task for object-oriented design. The intention is that the knowledge-based system should capture the know-how of an expert system designer and assist an inexperienced system designer to create good designs.

Key terms:
Object-oriented Design; Knowledge-based Support System; Design methodology; Object-orientation; Software Engineering; Software Development Life Cycle; Design Cycle; Object-oriented System Development; System Designer; Expert System.

Opgedra aan:

Johan, die Liefde in my lewe

Corlia, die Vreugde in my lewe

Pa en Ma, die Bron van Inspirasie in my lewe.

Acknowledgements

To my Lord and Saviour Jesus Christ - in whom all things are possible - Thank You for the opportunity to do this work.

A special word of thanks to my family, Johan and Corlia, who suffered all the way - I love you!

My sincere gratitude to my Father and Mother for all their hard work, loyalty, dedication and "We will help you wherever we can"-attitude.

Thank you to Izak and Lynnette, Anne-Mari, Ester and Johan for their encouragement during the years of study and to my Father-in-law and Mother-in-law for their interest in my progress.

A word of thanks to Professor Chris Bornman, Head of the Department of Computer Science and Information Systems, University of South Africa, for approving my sabbatical leave.

Thank you very much to Dr Andrew Whiter of Objective Solutions for his helpful attitude towards my work.

Finally, I gratefully acknowledge the guidance of my supervisor, Professor Lerine Steenkamp - my sincere thanks for helping me turn the byways into highways.

Table of Contents

Preface

This is a dissertation of limited scope (weight 5 modules) and reports on research done towards the MSc-degree in Information Systems (the MSc-degree in Information Systems has a weight of 10 modules).

Course work comprising five modules forms the other half of the MSc-degree:
INF417-N (Software Engineering) (weight: 1 module),
INF483-Y (Software Engineering Environments) (weight: 1 module),
COS452-H (Artificial Intelligence 2) (weight: 1 module),
One special topic project (weight: 2 modules) on *An Evaluation of an Application Development Methodology in a Fourth-Generation Environment.*

The investigation forms part of the **Object-oriented Information Systems Engineering Environment (OISEE)** project within the Department of Computer Science and Information Systems at the University of South Africa. The project is formulated in terms of a general framework of reference models that structures the technological foundation of information systems engineering into separate concerns. The following reference models were defined for the project:

- The [1]**Development Process Reference Model**
- The Quality Assurance Reference Model
- The Technology Reference Model
- The Target System Reference Model.

The **Development Process Reference Model** is concerned with the Information System Development Life Cycle, according to the following aspects:

- The Management Aspect
- The **Life Cycle Aspect**
- The **Methods Aspect**.

For the purpose of this research, an object-oriented spiral **life cycle** model is adopted, consisting of a Feasibility Cycle, an Analysis Cycle, a Design Cycle and an Implementation Cycle. This research concentrates on the **Design Cycle** within this life cycle model.

[1] The reference models and aspects in bold are relevant to this investigation.

List of Figures

List of Tables

List of Exhibits

List of Appendices

List of Abbreviations and Acronyms

AI	Artificial Intelligence
DDE	Dynamic Data Exchange
DLL	Dynamic Link Libraries
DFD	Data Flow Diagram
ES	Expert System
GI	Graphical Interface
KB	Knowledge Base
KBS	Knowledge-based System
KS	Knowledge System
OMT	Object-Modeling Technique
OOA	Object-oriented Analysis
OOD	Object-oriented Design
OOP	Object-oriented Programming
SE	Software Engineering
SEE	Software Engineering Environment
SDLC	Software Development Life Cycle

Definitions of Terms

CASE	-	The automation of the software engineering process by making use of computerized tools. These tools can be applied to the full or partial life-cycle.
DDE	-	Facility provided by Microsoft Windows to exchange data between various applications.
DLL	-	Facility provided by Microsoft Windows. Compiled code from various languages may be used interchangeably.
DFD	-	A graphical representation of the relationship between input and output values of processes.
Expert system	-	An expert system is a system which is an "expert" in some narrow problem area.
Knowledge Base	-	The repository of knowledge, represented as rules and facts that capture the reasoning criteria which a human expert applies to solve a problem in a particular area of knowledge; for a methodology it may contain the knowledge rules about the methodology and its standards.
Knowledge-based system	-	A knowledge-based system is a system in which problem domain knowledge (in the knowledge base) is explicit and separate from the general knowledge (in the inference engine).
Life cycle	-	A cyclic process used in system development. It usually takes on the form of analysis, design, implementation and maintenance.

Method	-	A regular, orderly, definite way of doing anything, applied to the specific approach to carrying out one or more of the software development activities; it is frequently based on an abstract or intellectual model of how to accomplish an activity; it is implemented by utilizing procedures and tools.
Methodology	-	A collection of methods and techniques which provides the overall approach to developing and improving software; usually based on an underlying intellectual model (the paradigm).
OMT	-	An object-oriented development methodology which uses object, dynamic and functional models throughout the life-cycle.
Paradigm	-	An abstract or intellectual model on which something is based.
Polymorphism	-	Derived from the Greek meaning: "Many shapes". The same definition is used for tasks that are implemented differently.
Software Development Life Cycle	-	a framework of orderly, interrelated activities which facilitate the development, implementation and maintenance of an information system.
Software Engineering	-	A systematic approach to systems development, making use of sound engineering principles and good management practice to obtain software of a high quality. This process is usually based on some form of life-cycle framework.

Software Engineering Environment	-	An environment which provides support for a particular development methodology and is also referred to as a Computer-Aided Software Engineering (CASE) environment. This support is accomplished through integrated CASE tools, utilities, procedures and one or more databases.
Target System	-	The system that is under development for the purpose of implementing it.
Technique	-	An informal method.
Tool	-	A mechanism for rendering a method executable; computer tools are the computer programs which make the execution or implementation of the steps of a method possible.

CHAPTER 1
Statement of the Problem

1.1 Introduction

Software Engineering (SE), first identified as a discipline in 1968, has two technical aspects: Software engineering-in-the-large, (at the level of software systems engineering with a number of developers) and software engineering-in-the-small, (at the level of the program and individual programmers). The ultimate aim of software engineering, in either of the two technical aspects, is to produce quality software systems efficiently. This may be achieved by well-established project management standards, a sound methodological approach, good engineering principles and reliable tools in support of all phases of the *Software Development*

Life Cycle. Project management includes planning project development, managing the workmanship and guaranteeing that the work is carried out to the required standards, on time and within budget (Sommerville, 1992). The purpose of a life cycle approach to software engineering is to enhance the productivity and quality of software systems. The software development life cycle typically includes four phases, namely feasibility, analysis, *design* and implementation.

In software development, disappointment with regard to *quality* and *productivity* is still an issue. The well-known structured software development approach has not fulfilled general expectations, namely producing high quality software within time and cost constraints. High quality software systems are systems which are maintainable, reliable and efficient, and which fulfil end-user requirements. It is claimed that an alternative paradigm, called *object-orientation*, may meet these expectations and, for this reason, the object-oriented approach to the design phase has been chosen for this investigation. The main focus of this research is to investigate support for *object-oriented design*, which in this case will be a *knowledge-based system*. The intention is that the knowledge-based system should capture the know-how of an expert or specialist system designer. Inexperienced system designers will then be able to use this support system to create good designs.

1.2 The Problem and its Relevance

The inexperienced system designer needs a support system to guide [1]him through

[1] The masculine form of the third person is used throughout to represent both genders.

the process of object-oriented design. The domain of discourse is growing increasingly complex, the design methodologies are sophisticated, and the current SE environments are tangled and advanced. This support system must be able to guide the system designer cautiously through the design process, according to a sound design methodology, highlight opportunities for prototyping (for example to develop a program for user experiment action), and to point out where to iterate (for example when quality assurance criteria have not been met).

1.3 Current Status of the Area of Investigation

When the complexity of software systems began to exceed the capabilities of the existing structured development techniques, attention was focused on the need for new methods. These methods must ensure that, during software development, high productivity is achieved for delivering reliable and maintainable systems with good quality. Software development has become very expensive because of high personnel costs and low productivity. The quality of software systems is poor because software performance is often unreliable (caused by the existence of undetected errors) and software maintenance is often complex and error-prone.

This section concentrates on key issues relevant to the area of investigation, namely: The design phase, object-oriented design, software process models and knowledge-based systems in support of design.

1.3.1 The Design Phase

The development of large software systems must be done in a well-defined way because it involves many different activities which are usually performed by a team of people. The correct system should be produced on time and within budget. For this reason the total development of the software, from conception to final delivery, is organised into one or another Software Development Life Cycle. There are different methodologies which organize this overall life cycle in different ways. A methodology involves various methods. These methods specify how the phases of the life cycle should be handled. Each method may implement specific techniques. Most software development methodologies support three basic phases of the software development life cycle, namely analysis, design and implementation.

Starting with the functional (behavioral) specifications which are the products of the analysis phase, the objective of design is to create a plan on which the actual building of the system will be based during the implementation phase. A design specifies the specific object modules which need to be written, and how the overall system will physically operate. The design process involves experience (because the design process is built upon innovative design ideas) and a large body of knowledge (consisting of principles, techniques and rules of thumb which a system designer requires to transform his innovative design ideas into working solutions).

The design phase is perhaps the most loosely defined since it is a process of gradual decomposition towards more and more detail. It is a creative process, a process of inventing a solution where none existed before.

There are several design approaches. These design approaches are tested ways of creating designs which have often proved to be good designs. However, design approaches do not take away the fact that design ideas have still to be created and judged in terms of criteria in order to establish whether or not a good design has indeed been achieved. One design approach is *Structured Design* (Colter, 1982). Here, the system designer produces a software solution to a problem in such a way that the solution has components and interrelationships which correspond to those of the problem. Another approach is the *Data-Driven Design* (Orr, 1971). The system designer determines the structure of data which best reflects the problem at hand. Then the system is designed on the basis of the structure of data. *Object-oriented Design* (Jackson, 1983) determines how interacting objects are structured into software sub-systems. An object, the key concept here, is a package containing data and associated procedures which operate on that data.

1.3.2 Object-oriented Design

Object-orientation provides a new paradigm for software construction. This new paradigm aims at achieving software reliability, efficient design of software, higher-quality software design and easier maintenance of software systems. In this new paradigm, *objects* and *classes* are the building blocks, while *methods*, *messages* and *inheritance* produce the primary mechanisms. *Objects* are "packages" which include both the data and the procedures which act upon the data. This packaging is referred to as *encapsulation*. The procedures which reside within the object take on a new name, i.e. *methods.* An object, on the other hand, may act and is activated by *messages* from other objects. Objects which have a common use and behavior are grouped together in a *class*, and new classes may be created which *inherit* the characteristics from classes already built, plus any special characteristics defined for that specific class. Thus new classes of objects may be defined from existing ones by simply defining how they differ from the originals. This feature enables the programmer to *re-use* existing classes and to program only the differences. The object-oriented paradigm offers a new level of abstraction, with prebuilt libraries of classes and even prebuilt application-specific class libraries or *frameworks.*

One of the main motivations and benefits of object-oriented development is the productivity gains which may be realized through re-use. If object-oriented analysis (OOA) and object-oriented design (OOD) components are developed and verified, and object-oriented programming (OOP) components are constructed and tested, and these components may be re-used in other applications, a fast and economical way of developing systems will be established.

It is not crucial to use an object-oriented approach in all of the phases of software development. For example, an object-oriented system design need not necessarily be implemented in an object-oriented programming language. However, by doing so, a cleaner conceptual mapping between the design and coding phases of a software project is provided (Atkins & Brown, 1991).

1.3.3 Software Process Model

The life cycle framework concept has been adopted from the engineering discipline. It is a well-phased framework in which the development of any product takes place. In particular, as far as software is concerned, the *software process model* is a conceptualization of the life cycle framework notion. A software process model (Du Plessis, 1992) steers the software development process, which means it guides the way in which the software is built from user requirements. The software process consists of a set of technical and management activities in which the software developer, the software manager and the end-user participate.

It is now important that frames of reference should exist that establish shared understanding among participants so that development may benefit (Du Plessis, 1992). A number of viewpoints, or aspects, concerned with the software process model are represented by different reference models. These reference models are: A Target System Reference Model, a Technology Reference Model, a Quality Assurance Reference Model and a Development Process (DP) Reference Model. Software development includes the modeling of the characteristics and behavior of an application, the target system. One of the results of the modeling activity is a conceptual *Target System Reference Model.* A *Technology Reference Model* structures the development environment within which an application is developed and a *Quality Assurance Reference Model* is concerned with the quality of process and product. The *DP Reference Model* guides the set of technical and management activities which takes place. It is concerned with a particular software development life cycle (SDLC), according to which the *management* *aspect*, the *life cycle aspect* and the *methods aspect* may be viewed. For the OISEE project the aspects have been interpreted as follows:

(i) ***The Management Aspect*** : This aspect enables members of the project and development management team to view a project at three levels of abstraction. These levels are the *Universal Level*, the *Worldly Level* and the *Atomic Level*.

(ii) ***The Life Cycle Aspect*** : Originally Boehm (1986) proposed a spiral life cycle model. Du Plessis & Van der Walt (1992) explain a revised spiral model for object-oriented development. The development is cyclic, where the cycles are as follows: The Feasibility Cycle, the Analysis Cycle, the Design Cycle and the Implementation Cycle. Each cycle is characterised by four quadrants, namely *Issue Formulation*, *Analysis and Evaluation of Alternatives*, *Development* and *Review/Planning*.

(iii) ***The Methods Aspect*** : The technical development process and the related management tasks were guided by a chosen set of object-oriented methods.

1.3.4 Knowledge-based Systems in support of Design

We may now ask which knowledge-based software support systems would be useful to a system designer of an object-oriented application.

Current applications of Artificial Intelligence (AI) fall into the following categories: Knowledge-based systems, natural language processing, speech understanding, robotics, and image and pattern understanding. The terms knowledge-based systems (KBS), knowledge systems (KS) or expert systems (ES) are used for programs which model the experience of one or more people to help the user make decisions.

Rolland & Proix (1986) argue that the design process is a complex, iterative, lengthy and monotonous task which is characterized by a measure of uncertainty

(i) ***in the definition of the problem***, because the boundaries of the application domain are very seldom clearly defined, and the goals and scope of the system are generally fuzzy.

(ii) ***in the manner of choosing an information system conceptual schema***, because the same application domain may be described by different schemata.

(iii) ***in the manner of translating the conceptual schema into a physical schema***, because this mapping is dependent on both the technical environment which is available and on the end-user needs.

Owing to this uncertainty, a purely algorithmic solution is impossible. A system designer may control the design process because he uses formal techniques and experimental rules simultaneously. He continuously uses his experience which allows him to recognize typical situations, to resolve problems by comparison and to know when to iterate or prototype. From this it may be seen that, although the design is a creative process, one may still learn from an expert about the typical cases and pitfalls.

1.4 Proposed Solution

The purpose of the investigation is to support the software development process, in particular the *design* of the *software development life cycle* in an *object-oriented environment*, with a *knowledge-based system.* The support will be in accordance with a selected methodology. The investigation also aims to construct a prototype which will partially automate the work of the system designer. These were the essential issues which guided the research.

1.4.1 Method of Investigation

The investigation started with a literature study concerning the identified issues of the problem domain. An analysis of relevant references followed, which was both interpretive and evaluative. The *Design Cycle* in the *revised spiral software development life cycle* was the focus in the investigation. The *object-oriented paradigm* was chosen as the basis for development and the *Object-Modeling Technique (OMT) methodology* (Rumbaugh *et.al.*, 1991) was adopted. A number of knowledge-based environments were evaluated according to a set of criteria. The set of criteria used, are grouped into eight categories, namely: End-user interface criteria, developer interface criteria, system interface criteria, inference engine criteria, knowledge base (KB) criteria, data inference criteria, cost-related criteria and vendor-related criteria. The *knowledge-based environment* which was chosen, is [2]*Kappa-PC.* An analysis of the literature made it possible to postulate a *hypothesis*, make certain *assumptions* and decide on the *constraints* for the investigation. A proposed solution was *conceptualised*, based on an analysis of the design task and a prototype was built to *demonstrate the concept.* This prototype was *evaluated* according to criteria which were synthesized during the investigation. The set of criteria concentrates on good

[2] Kappa-PC is a registered trademark of IntelliCorp, Inc.

object-oriented design principles, a sound design method and the support which the knowledge-based environment gives. The investigation concluded with the validation of the original hypothesis.

(i) The Hypothesis

The investigation is based on the hypothesis that it is possible to create an aid for inexperienced system designers in a software development process, namely a knowledge-based workbench which supports object-oriented design as illustrated in the following block diagram (*Figure 1.1*).

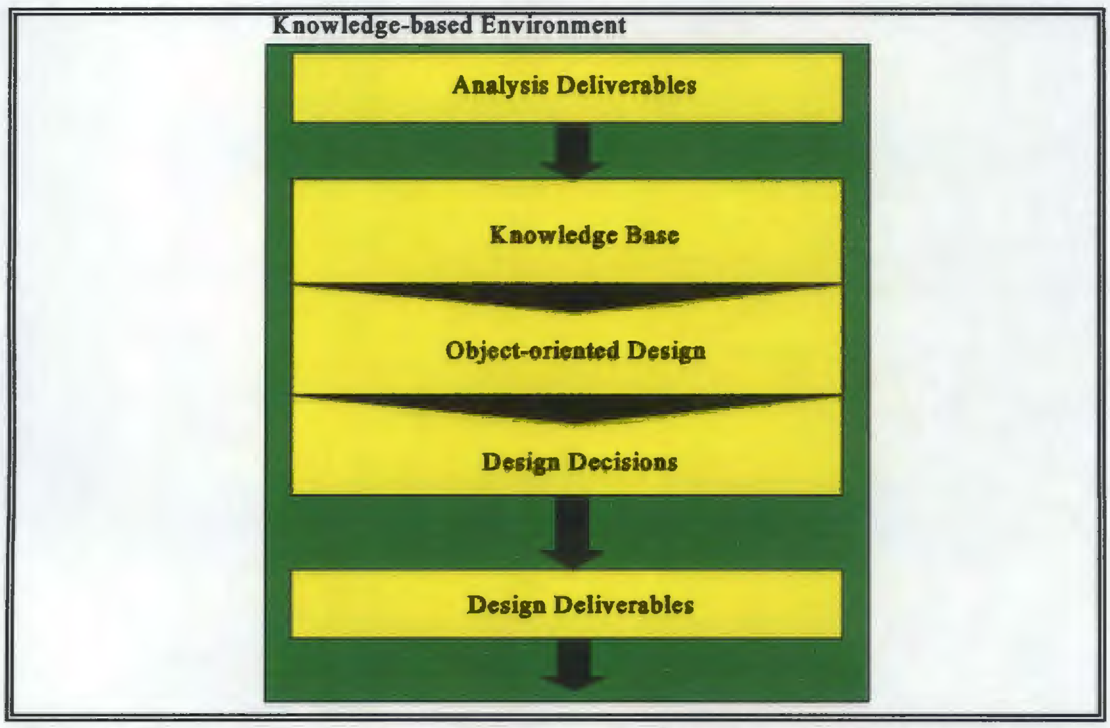


Figure 1.1 Conceptualization of a knowledge-based workbench which supports Object-oriented Design

(ii) ***The Assumptions***

The Analysis Cycle has been completed and the analysis deliverables, which form the point of departure for this investigation, are available. The analysis deliverables include the analysis part of the repository (the format of which is determined by the meta model of the OMT methodology), and the analysis document (the Requirements Specification Document). The object-oriented paradigm is followed for design and the revised spiral model for object-oriented development is adopted (Du Plessis & Van der Walt, 1992).

(iii) ***The Constraints***

Constraints for the investigation include the following:

- A personal computer (PC) environment
- The Design Cycle of the revised spiral model, within the parameters of the OISEE project in the Department of Computer Science and Information Systems at UNISA.
- The application domain includes functional transformation systems (e.g. batch computation and continuous transformation systems), time-dependent systems (e.g. interactive interfaces and dynamic simulation), and database systems (e.g. transaction managers).
- The scope of the research was to meet the requirements for a partial dissertation.

Several relevant issues are mentioned but fall outside the scope of the investigation. They are:

- Project management for a design team
- The role of the end-user in the Design Cycle
- Implementation, planning and cost estimation

- Estimating the cost of the design process
- The coding of application programs
- Prototyping during the Design Cycle
- Re-usable components in the Design Cycle
- Quality assurance and verification of the Design Cycle
- Consistency and completeness of the Design Cycle.

(iv) ***Literature Survey***

The identified issues guided the literature study, during which references were analytically reviewed, interpreted and evaluated for significance in terms of the hypothesis and aims of the investigation.

(v) ***Conceptualisation***

A synthesis is made of ideas concerning the relevant knowledge required for the tasks of object-oriented design to formulate a conceptual model as a proposed solution to the problem of supporting design steps by means of a knowledge-based environment.

(vi) ***Demonstration of Concept***

The conceptual model was prototyped within a knowledge-based environment. The domain of discourse is the design process when designing an application.

(vii) ***Evaluation***

An evaluation of the prototype against established criteria follows.

(viii) ***Conclusion***

Based on this evaluation, the hypothesis and assumptions of the investigation were validated.

1.5 Structure of the Dissertation

The dissertation consists of seven chapters, followed by exhibits and appendices.

Chapter 1 identifies the research areas which are relevant to the investigation. A motivation for investigating this area of research is given. The particular aspects which will be considered are stated. A possible solution is proposed for dealing with the problems of constructing a good design in an object-oriented environment. This is followed by the method of investigation which guided the research. The chapter concludes with an overview of the content of the dissertation.

In Chapter 2 an overview of the design process is given. The software process model is explained and the importance of multi-perspectives is discussed. The categories of structured design methods are described, namely top-down structured design and data-driven design. The principles of object-orientation and the object-oriented design process are explained and object-oriented design methods are reviewed. Good design principles are underlined and knowledge-based support for object-oriented design is briefly discussed. The chapter is concluded with a summary.

Chapter 3 reports on knowledge-based systems in general, and expert systems in particular. The structure of an expert system, the main players in an expert system and the basic characteristics of an expert system development environment are discussed. Knowledge acquisition, knowledge representation and inferencing are explained, and the selection criteria for an expert system development environment are categorized and explained. Kappa-PC is evaluated against these criteria and a summary of the chapter, as well as conclusions made in the chapter, follow.

Chapter 4 is devoted to the design method which was identified during the investigation, namely OMT (Rumbaugh *et al.*, 1991). Modeling in general is discussed and the OMT, in particular, is explained under the following headings: The object model, the dynamic model, and the functional model. A summary of all these models is given. The OMT method for design is discussed in detail with reference to System Design and Object Design. The organization of the design knowledge base (KB) is explained and the conceptual model of the proposed solution is illustrated. A summary and conclusions end the chapter.

Chapter 5 is a description of the knowledge-based environment which was identified during the investigation, namely Kappa-PC. The different key concepts in Kappa-PC are mentioned and the Kappa-PC building blocks are explained. The KAL language, the end-user interface, the developer's interface, the external data sources interface together with the programming languages interface, and the knowledge base (KB), with its rule-based reasoning, are discussed.

In Chapter 6 the purpose and scope of the design prototype is explained. The purpose and format of the User's Manual, for the prototype which is built to serve as a demonstration of concept, are given. The complete manual is included as *Appendix F* and an explanation of the demonstration is given in *Appendix G.* The source code of the design prototype is in *Appendix H.*

The contribution this research makes is evaluated, namely whether or not the prototype which was built serves adequately to demonstrate the concept. The research results are evaluated and summarized and the conclusions drawn during the investigation are stated in Chapter 7. Areas for further investigation are proposed.

CHAPTER 2
The Design Process

CONTENTS

2.1 Introduction

Design is the development of a model of the internal structure of the system. It is a blueprint of *how* the system should be constructed in order to display the behavior of the system (i.e. a description of *what* the system must do to meet the needs of the users) as it was modeled in the Analysis Cycle. Design is a creative activity because, although there are well-established methods which guide design, the essence is to create a solution where none existed before. The success of the design process depends on original ideas and this is the reason why experience is a very important factor that makes for a good system designer. Although talent and experience are involved, one may not ignore the large body of knowledge, consisting of principles, techniques and rules of thumb, which a system designer needs, in order to help him create a working solution.

2.2 Software Process Model

The software process is the way in which the software is built, starting with user requirements. The software process architecture guides the software process. A specific instance of a software process architecture is referred to as a software process model (Humphrey, 1989).

For the OISEE project, and hence for this investigation, Humphrey's (1989) three-level software process model was adopted for the *Management Aspect*. This model consists of a Universal Level, a Worldly Level, and an Atomic Level. The *Universal Level* provides a global view of a software system project for senior management. The global view may be structured by means of a software development life cycle (SDLC) framework which guides the project. The next level down is the *Worldly Level* which guides the sequence of development and management tasks of the cycles of the SDLC. The orderly and prescriptive manner of performing the tasks of the Worldly Level is detailed in the *Atomic Level*, for junior management. The DesignNet Model, proposed by Liu and Horowitz (1989), will be the representation scheme for picturing tasks or activities, deliverables and status reporting, on all levels of the three-level model. The representation scheme was derived from AND/OR graphs and Petri nets. DesignNet conveys information regarding the schedule, the work-breakdown-structure, manpower allocation, costing and current status of the project on all three levels. The principle advantage of using DesignNet is that all participants involved in managing a project share information and may communicate across project levels. The Universal Level of the development process model is visually depicted in the DesignNet notation according to the revised spiral model as seen in *Figure 2.1*.

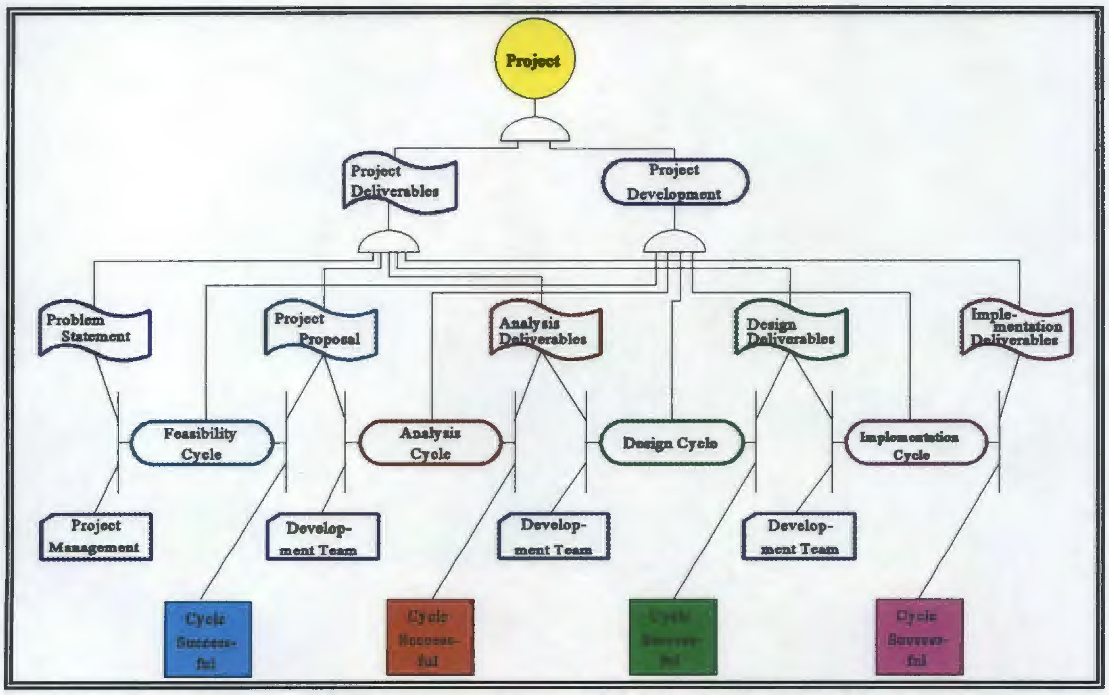


Figure 2.1 Universal Level of the Development Process Model (Du Plessis & Van der Walt, 1992)

For the *Life-Cycle Aspect*, the spiral model (Boehm, 1986) was considered for the OISEE project. Boehm took the system development life cycle and introduced a risk-driven approach into the development of software products, calling it the "spiral model". Whereas the classical waterfall model (Royce, 1970) was a specifications-driven model with prototyping sometimes included, the spiral model also calls for an evaluation of the risk of all the products developed during the previous cycle of the spiral, including the plans for the next cycle and the resources required to carry them out. After the completion of a cycle of the spiral model, an evaluation is made as to whether to continue or abort the development process. If the project should continue after such a risk evaluation, the next cycle of the spiral model is started. Otherwise, the development stops and an evaluation of the entire project is made. The advantages of the spiral model are the concept of risk assessment and risk management which are introduced into the system development process. With the spiral model, iterative processes are possible (Sage & Palmer, 1990). For the purpose of this investigation, a revised spiral model for object-oriented development, as proposed by Du Plessis and Van der Walt (1992), is adopted. This model, as shown in *Figure 2.2*, has four quadrants, namely Quadrant 1 - Issue Formulation; Quadrant 2 - Analysis and Evaluation of Alternatives; Quadrant 3 - Development; Quadrant 4 - Review/Planning.

For object-orientation four cycles are identified, as seen in *Figure 2.3*, namely Cycle 1 - Feasibility; Cycle 2 - Analysis; Cycle 3 - Design; Cycle 4 - Implementation. This research will concentrate on, and refine, the Design Cycle.

For the *Methods Aspect* a set of object-oriented methods were chosen to guide the technical development process and the associated management tasks.

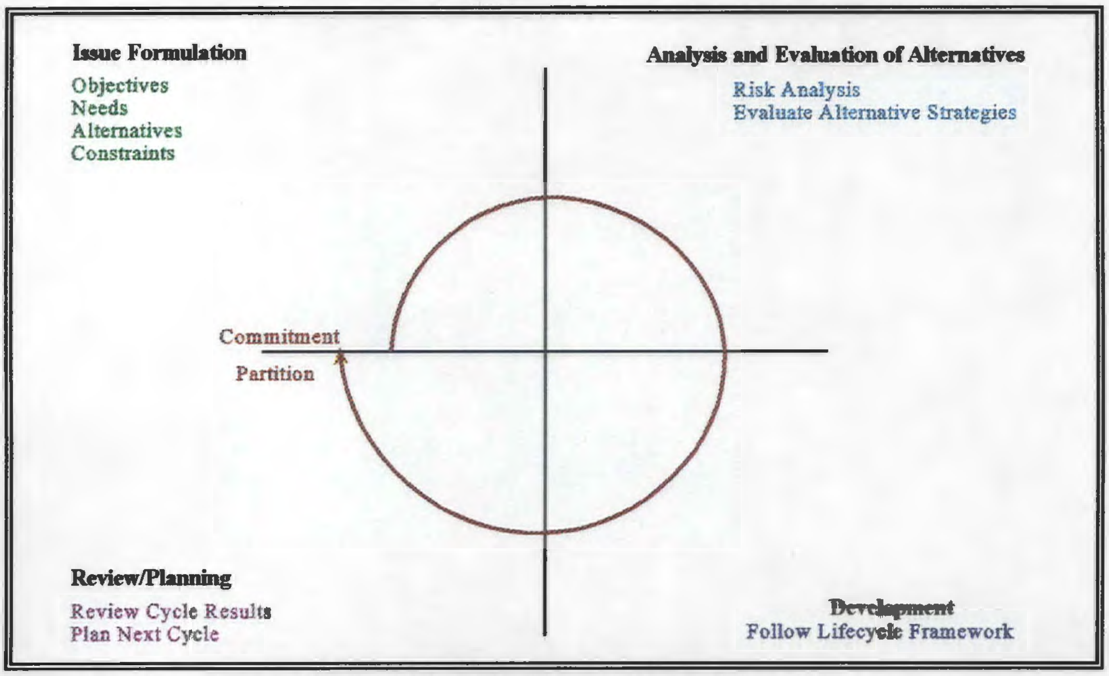


Figure 2.2 The quadrants of the Spiral Model (Du Plessis & Van der Walt, 1992)

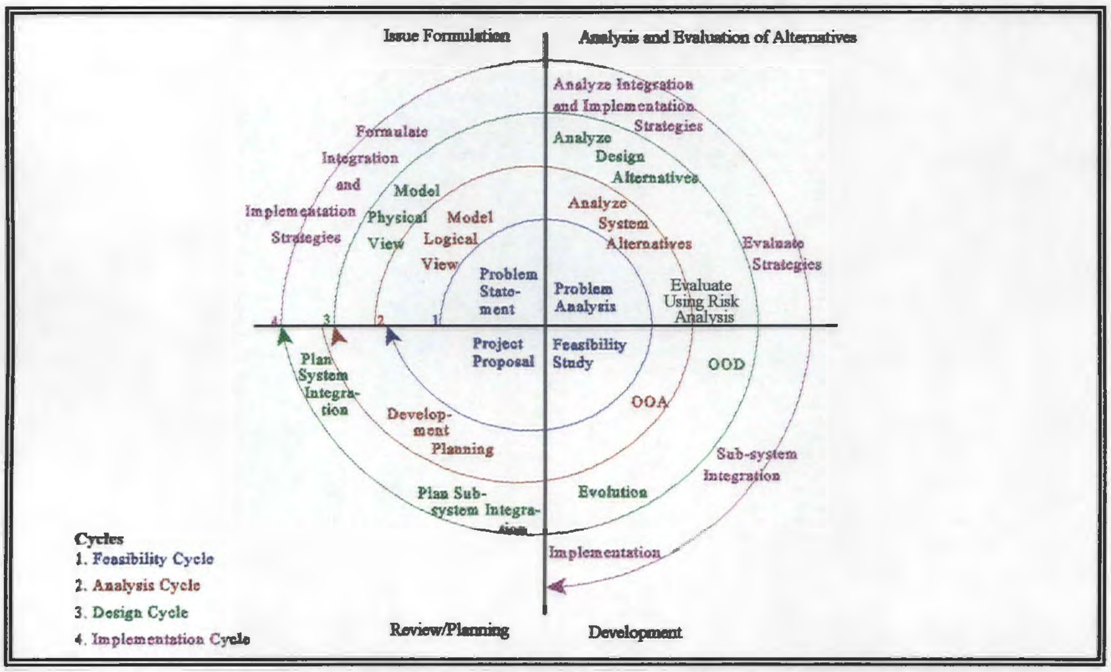


Figure 2.3 Revised Spiral Model for object-oriented development (Du Plessis & Van der Walt, 1992)

The combination of Humphrey's (1989) software process model, the revised spiral model for object-oriented development (Du Plessis & Van der Walt, 1992), and the DesignNet model (Liu & Horowitz, 1989) was proposed by Du Plessis and Van der Walt (1992). This specific combination is adopted for this investigation.

2.3 Multi-Perspectives

Different Information System Methodologies exist where each emphasizes certain perspectives. There are three essential perspectives and most methodologies emphasize one perspective to the exclusion of the other two. The three perspectives are (Olle *et al.*, 1988):

(i) Data-oriented,
(ii) process-oriented, and
(iii) behavior-oriented.

(i) The ***data-oriented*** perspective stresses a comprehensive and precise analysis of the data and its relationships. The emphasis is on retrievability of all information, independently of storage representation, resulting in the expression of the integrity restrictions which the data must satisfy.

(ii) The ***process-oriented*** perspective is the oldest perspective. It started when the computer was regarded as a convenient tool for performing specific processes, such as generating a payroll. A trend towards moving away from the computerizable process followed. The emphasis then shifted towards an analysis of the activities as performed in business. The belief

was that these activities could beneficially be computerized.

(iii) The ***behavior-oriented*** perspective focuses on the dynamic nature of the data. The need to analyze and understand events in the real world, which may have an impact on data recorded in the information system, stresses a dynamic view of the business area and of the information system. The concentration is on changes over time, changes which may take place and changes which are observed to take place.

2.4 Structured Design Methods

Structured Design may be seen as the development of a plan of a computer system solution to a problem which has the same elements and interrelationships among the elements as the original problem (Page-Jones, 1988). In this section two categories of design methods are discussed. These are Top-down structured design and Data-driven design.

2.4.1 Top-down Structured Design

People have realised that the ability to manage the system development process is not sufficient for the needs of the increasingly complex systems. During 1968, at a NATO sponsored conference, Dijkstra (1969) talked about a structured approach for the first time. He demonstrated his idea by making use of his now well known control constructs used during the Implementation Cycle, namely sequence, choice and iteration as illustrated in *Figure 2.4.* He argued that the flow of control should follow one of these forms. Parnas (1972) proposed the idea of partitioning a system into modules because the problem-solving notion of divide-and-conquer permits one to subdivide a difficult problem into sub-problems repeatedly until the resulting problems become manageable. In top-down structured design, these subproblems are called "modules". The top-down structured design method is a functional method. Structured design has five chief goals (Page-Jones, 1988):

- Letting the nature of the problem guide the nature of the solution.
- Reducing system complexity by partitioning a system into hierarchies of modules.
- Using graphical representation schemas to render systems more understandable, e.g. structure charts, and supporting these by means of pseudocode.
- Offering a set of strategies for developing a design solution from a well-defined statement of a problem.
- Providing a set of criteria for evaluating the quality of a given design.

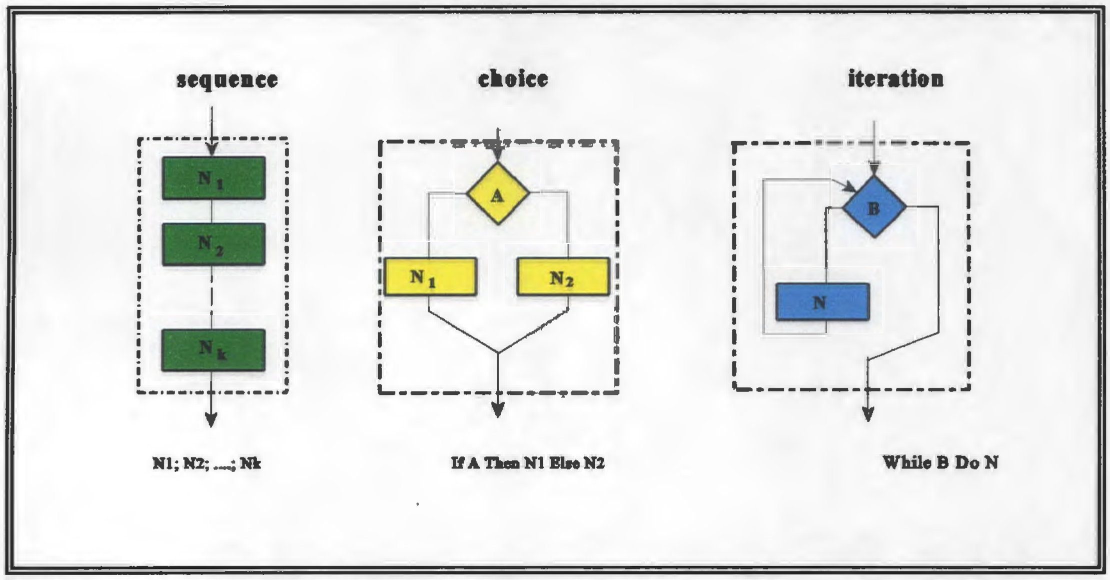


Figure 2.4 Dijkstra's Control Constructs

Structured Design is a disciplined approach to computer software design, based on established design principles with the following advantages:

- A system may be divided into *partitions or modules.*
- It may be made *verifiable.*
- The *understandability* of a system is better because of the notion of modules.
- *Communication* between people is better because the understandability of the system is better.
- *Modifiability* is easier because the understandability is better.
- *Re-usability* is better because a module with functional cohesion may generally be re-used in other contexts.

2.4.2 Data-Driven Design

The data-driven design is best illustrated by the work of Jackson (1975 and 1983) and the methods of Warnier and Orr (Orr, 1971). In this method, the structure of a software system is obtained from mapping system inputs to outputs. Data-driven designs have been successfully applied, in particular to information management systems.

2.5 Object-oriented Design

Object-oriented design aims at creating quality designs which adhere to good design principles and which may be efficiently implemented in a suitable implementation language. Object-orientation has its roots in the principles behind the SIMULA programming language. The emphasis of research in object-orientation has been on implementation aspects, such as the development and use of object-oriented programming languages. The potential benefits of object-orientation for the analysis and design of software systems have not been recognized until recently (Van de Weg & Engmann, 1992).

Jackson's work is regarded as a forerunner of object-orientation. He is the father of a system development method called *Jackson System Development (JSD)* (Jackson, 1983). In this method the real world is described in terms of entities, actions they perform or suffer, and the orderings of those actions. For example, in a bank the entities are the *customers*; the actions are *invest*, *withdraw*, *deposit* and *terminate*; the ordering of the actions is *invest* first, then a number of *withdraw* and *deposit* actions, then finally, *terminate*. An entity exists as part of the real world outside the system, it performs or suffers actions in a time ordering, it is capable of being regarded as an individual, it may be uniquely named, and the system must be required to produce or use information about it whereas an action takes place at a particular point in time and cannot be extended over a period (Connor, 1985). For example, to *sleep* is not an action, but to *wake up* is indeed an action.

This description by Jackson of entities, actions they perform and the ordering of such actions is the essence of the object-orientation paradigm.

2.5.1 Principles of Object-orientation

Object-oriented development is an approach to software development in which the decomposition of a system is based upon the idea of an object. An *object* is an entity, the behavior of which is characterized by the actions (operations) which it suffers (this means it is acceptable that the action may be performed upon the object) and by the actions which it requires of other objects (Booch, 1987). Thus object-orientation is an approach which exploits encapsulation or "packaging" in the process of designing and building software. The object-oriented paradigm, at its simplest, takes the components of a software system, namely data and procedures, and de-emphasizes the procedures, stressing instead the encapsulation of data and procedural features together. The encapsulation of data and related procedural features, forms an object. *Figure 2.5* demonstrates an object. Any interaction with an object is done by sending a message to the object. This means using one of the procedures which the object makes available for interacting with its internal state (data).

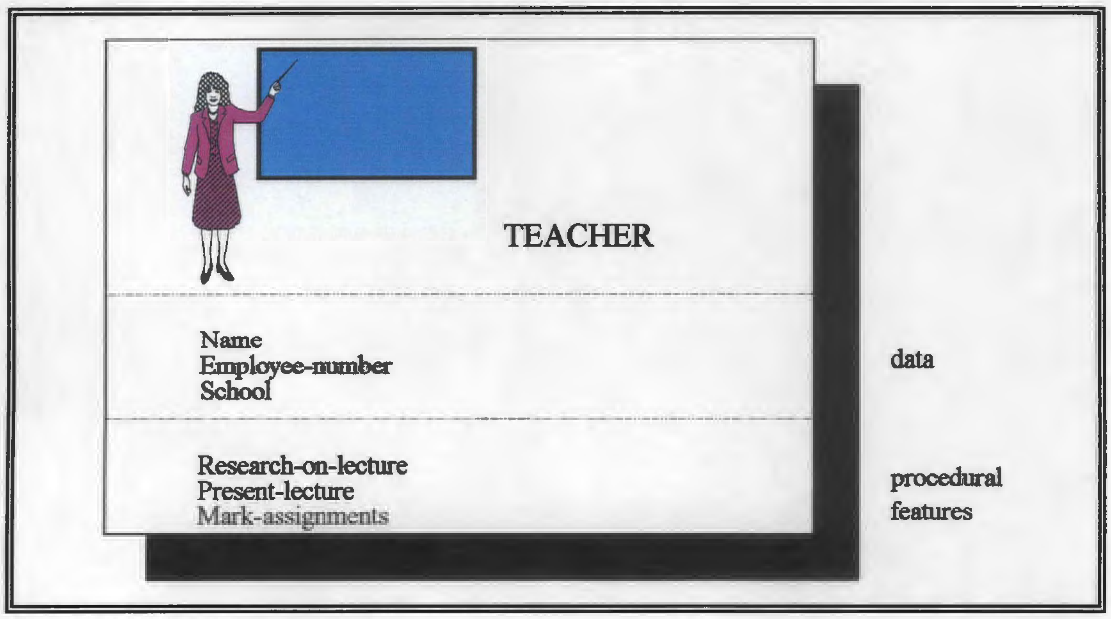


Figure 2.5 TEACHER as an object

The claims which are made about object-orientation are that:

- It is more natural to think in terms of objects.
- The model of the problem space fits more directly into the solution space.

The elements which underlie the object-oriented technology are not unique to object-oriented systems, but they are particularly well supported in object-oriented systems. They are:

(i) Identity
(ii) Classification
(iii) Polymorphism
(iv) Inheritance
(v) Synergy
(vi) Abstraction
(vii) Encapsulation
(viii) Information-hiding
(ix) Modularity
(x) Hierarchy
(xi) Combining data and behavior
(xii) Sharing
(xiii) Emphasis on object structure, not procedure structure
(xiv) Typing
(xv) Concurrency
(xvi) Persistence.

(i) ***Identity*** is the nature of an object which distinguishes it from all other

objects. When data is grouped into separate entities, called objects, each object has its own natural identity. This implies that two objects are different even if all their attribute values are identical. For example, John has exactly the same car (object) as Peter, the same model, the same features and even the same color, but the cars (objects) have different identities because the one car (object) belongs to John (class) and the other car (object) belongs to Peter (class). An *attribute* is a data value held by each object of a class, for example *model, features* and *color* are attributes of *car* objects.

(ii) ***Classification*** exist when objects with the same behavior (operations) and data structure (attributes) are grouped together into a class. An *operation* is an action or transformation which an object performs or is subject to. From the viewpoint of a class, each class is a definition of data and procedures that each instance of that class will contain, accordingly defining each instance's behavior. A class is thus a generalization of the characteristics and behavior of the objects belonging to the class. A class may be seen as an abstraction which only describes properties important to an application and ignores the rest. A given class usually has two kinds of clients, namely instances and subclasses (Micallef, 1988). Each object may be seen as an *instance* of its class. A class which inherits from one or more classes is called a *subclass.*

(iii) ***Polymorphism*** refers to the same operation behaving differently when applied to different classes, for example the *display* operation may behave differently when applied to the *text* as opposed to the *figure* class. The implication of polymorphism is that operations may be

defined at the class level and implemented for subclasses or objects by means of various methods. This means that a new method may easily be added when required. A *method* is a specific implementation of an operation by a certain class and a method is part of an object. A method is invoked by sending a message to the object instance of the class. A *message* (in [1]Smalltalk-80) is the activating of an operation on an object, containing an operation name and a list of argument values.

(iv) ***Inheritance*** is a powerful feature of object classes and is based on a hierarchical relationship between classes. Inheritance refers to the sharing of attributes and operations among classes in this relationship. A class may be refined into consecutive finer subclasses. Each subclass merges, or inherits, all of the properties of its superclass and adds its own unique properties.

(v) ***Synergy*** is the compilation of ideas. As regards object-orientation, this means that identity, classification, polymorphism and inheritance together complement each other synergistically. These aspects exist in isolation and characterize mainstream object-oriented languages. According to Thomas (1989), these various features come together to create a different style of programming.

(vi) ***Abstraction*** is ignoring an entity's unexpected characteristics, for example deciding how an object should be implemented, and concentrating on the essential, natural aspects of an entity, for example what an object is and does. Abstraction refers to a data structure together with its operations. Data abstraction applies to the data structure and procedural abstraction applies to the operations. The implication of

[1] Smalltalk-80 is a trademark of ParcPlace Systems.

abstraction is that one may view concrete and abstract things, and their relevant operations, as a modeling primitive.

(vii) ***Encapsulation*** is the grouping of both data and operations affecting that data, into a single object. Encapsulation separates the external aspects of an object from the internal implementation details of the object. This prevents a program from becoming so interdependent that a small change has enormous ripple effects. The ideal is that the implementation of an object may be changed without affecting the applications which use it. Combining data structure and behavior in a single entity, as claimed by object-orientation, makes encapsulation neater and more robust than in conventional languages which separate data structure and behavior.

(viii) ***Information-hiding*** allows one to remove from view some portion of those things which have been encapsulated by the object. Encapsulation draws a capsule around related things, which is then called an object. Information-hiding underlines that an object has a public interface and a private representation.

(ix) ***Modularity*** is the characteristic of a system which has been decomposed into a set of strongly cohesive and loosely coupled modules (Booch, 1991).

(x) ***Hierarchy*** is a grading or classifying of abstractions. A set of abstractions often forms a hierarchy. By identifying these hierarchies in a design, one may greatly simplify the understanding of the problem (Booch, 1991).

(xi) ***Combining data and behavior*** means that the caller of an operation need not consider how many implementations of a given operation exist. The

burden of deciding what implementation to use shifts from the calling code to the class hierarchy because of operator polymorphism. For example, invoking the *draw* operation on some figures implies that the decision on which procedure to use, circle or polygon, is made implicitly by each object, based on its class, whereas in a non-object-oriented environment, code must first distinguish the type of the figure and then call the appropriate procedure to display it.

(xii) ***Sharing*** is promoted at several different levels by object-oriented techniques. The sharing of code using inheritance is one of the main advantages of object-oriented languages. Object-oriented development also offers the prospect of re-using designs and code on future projects because of features such as abstraction, encapsulation and inheritance.

(xiii) ***Emphasis on object structure rather than procedure structure*** has the result that the emphasis falls on what an object *is*, rather than how it is *used*, according to Booch (1986) who said that software systems built on object structure are more stable in the long run.

(xiv) ***Typing*** is the administering of the class of an object. This administering prevents objects of different types from being interchanged or allows them to be interchanged only in very limited ways (Booch, 1991).

(xv) ***Concurrency*** allows different objects to act at the same time. This refers to tasks, activities or events whose execution may overlap in time.

(xvi) ***Persistence*** is the characteristic of an object by which its existence exceeds time and/or space. Existence exceeding *time* occurs when the object continues to exist after its creator ceases to exist. Existence exceeding *space* occurs when the object's location moves away from the address space in which it was created.

At this stage it seems correct to agree with Atkins and Brown (1991) when they claim that the primary benefit of the object-oriented approach is that it directly supports many of the good practices and goals of software engineering.

2.5.2 The Object-oriented Design Process

During the Analysis Cycle the focus is on *what* is to be done. System Analysts must first understand the problem domain at hand and the system's responsibilities within that problem domain. A complete *Problem Statement* is compiled. Then the conceptual entities or objects in the problem under analysis are modeled. Next, the interaction of the objects is modeled and lastly the processing in the problem is modeled (Shlaer & Mellor, 1988). After the modeling, an *Analysis Document* (or Software Requirements Specification) is compiled.

"*Object-oriented design is the method which leads to software architectures based on the objects every system or sub-system manipulates (rather than 'the' function it is meant to ensure).*" (Meyer, 1988). During the Design Cycle the focus changes to *how* it should be done. System Design consist of establishing a high-level strategy for solving the problem and constructing a solution. It includes making decisions about the organization of the system into subsystems, the allocation of sub-systems to hardware and software components, and conceptual and policy decisions which form the basis for detailed design.

The System Designer starts with the *Analysis Document* which consists of:

- A Problem Statement.

- A model of the static structure of a system which shows the objects in the system, relationships between the objects, and the attributes and operations which characterize each class of objects. This model answers the question: *What happens to it?*
- A model of those aspects of a system that are concerned with time and changes. Describing the flow of control, in other words the sequences of operations which occur in response to external stimuli, without considering what the operations do, what they operate on, or how they are implemented. This model answers the question: *When does it happen?*
- A model of the computations within a system. This model shows how output values in a computation are obtained from input values, without regard for the order in which the values are computed. This model answers the question: *What happens?*

Object-oriented design is an incremental process - the identification of new classes and objects usually results in refining and improving upon the semantics of existing classes and objects, and refining and improving upon the relationships among existing classes and objects.

Object-oriented design is also an iterative process - implementing classes and objects may lead to the discovery or invention of new classes and objects whose existence simplifies and generalizes the design.

According to Booch (1991), the process of object-oriented design generally tracks the following order of events:

(i) Identify the classes and objects at a given level of abstraction.

(ii) Identify the semantics of these classes and objects.

(iii) Identify the relationships among these classes and objects.

(iv) Implement these classes and objects.

(i) ***Identify the classes and objects at a given level of abstraction.***

Here two activities are of importance, namely:

- The *discovery of the key abstractions* in the problem space (the significant classes and objects), and
- the *invention of the important mechanisms*, which are the object structure that shows how different objects work together to accomplish some function.

(ii) ***Identify the semantics of these classes and objects.***

Establish the meanings of the classes and objects from the previous step. Identify the things which may be done to each instance of a class and the things which each object may do to another object. This identification may be done by viewing each class from the perspective of its interface.

(iii) ***Identify the relationships among these classes and objects.***

Establish how things interact within the system. With the *key abstractions* one must establish the use, inheritance, and other kinds of relationships among classes. As far as the *objects* are concerned one must establish the static and dynamic semantics of each mechanism.

(iv) ***Implement these classes and objects.***

This step involves two activities:

- Making design decisions affecting the representation of the classes and objects which were invented, and

- allocating classes and objects to modules, and programs to processors.

At this stage an inside view of each class and module is taken, to decide how its behavior should be implemented. This is not necessarily the last step in the design process because when completing this step, it is necessary most of the time to repeat the entire process, this time at a lower level of abstraction.

During *System Design* (or Preliminary Design), which is the first design task, the basic approach to solving the problem is selected. The overall structure, style and organization of the system, which is the *system architecture*, is decided upon. At the end of System Design, the *System Design Document* is produced which describes the structure of the basic architecture for the system as well as high level strategy decisions. After the System Design, the System Designer must start on the *Object Design* (or Detailed Design) during which the System Designer elaborates on the analysis models and provides a detailed basis for implementation. Object-oriented design ends:

- Whenever there are no new key abstractions (the significant classes and objects) or mechanisms (which provide the performance required of objects which operate together to accomplish some function), or
- when the classes or objects already discovered may be implemented by creating them from existing re-usable software components.

A *Design Document* is constructed after the object design task.

2.5.3 Object-oriented Analysis and Design Methods

Relatively little has been published on object-oriented methodologies for software engineering, but a few will nevertheless be reviewed. Some Object-oriented Analysis methods are included in this review for the sake of completeness.

Shlaer and Mellor (1988) describe a total *methodology for object-oriented analysis* which breaks analysis down into three tasks: Static modeling of objects, dynamic modeling of states and events, and functional modeling. Shlaer and Mellor say that their methodology is an approach to analysis only.

Coad and Yourdon (1990) also present an approach to *object-oriented analysis* which is similar to the original Object-Modeling Technique (OMT) as reported by Loomis, Shah and Rumbaugh (1987).

The *Object-Modeling Technique (OMT)* (Rumbaugh *et al.*, 1991) is a methodology which describes classes and relationships throughout the life cycle, based on the use of an object-oriented notation. In order to be able to describe all aspects of a system, the Object Model is enlarged by adding a Dynamic Model and a Functional Model. During the analysis task, a model of what the system is supposed to do is developed, regardless of how it is implemented. During the design task, the Object Model, Dynamic Model and Functional Model are optimized, refined and extended until they are detailed enough for implementation.

Booch (1986) describes the foundation of object-oriented software development.

He claims that Object-oriented software components model a person's perception of reality very closely. *Booch's Methodology* (Booch, 1991) consists of a collection of models which address the object, dynamic and functional aspects of a software system. Associations are mentioned but not incorporated into Booch's methodology.

2.6 Good Design Principles

The system designer is concerned with achieving the design objectives specified in the user implementation model, as well as with the overall quality of the design. The nature and quality of the design created by the system designer affect the ability of the programmers to implement a high-quality, error-free system. This also affects the ability of the maintenance programmers to make changes to the system after it has been put into operation. Ingalls (1981) suggests that "...... *a system should be built with a minimum set of unchangeable parts; those parts should be as general as possible; and all parts of the system should be held in a uniform framework*". Classes and objects are the key abstractions of the system when working with object-oriented design. How does one know if a given class or object is well designed? Booch (1991) suggests that there are five meaningful principles:

(i) Cohesion within a class or object.

(ii) Coupling between classes or objects.

(iii) Classes should be sufficient.

(iv) Classes should be complete.

(v) Classes should be primitive.

(i) ***Cohesion*** is the degree of interaction within a class or object, which refers to how the activities within a single class or object are related to one another. There are various levels of cohesion, of which functional cohesion is the most desirable. A functionally cohesive class or object performs only one problem-related task. Informational cohesion is also good. An informationally cohesive class or object performs a number of actions on the same data structure, with independent code for each action. For a good design, classes or objects must have a *high level of cohesion*.

(ii) ***Coupling*** is the degree of interaction between two classes or objects. The ideal is to make classes or objects as independent as possible. There are various levels of coupling, of which data coupling is the most desirable. Data coupling is coupling by elementary parameters where every parameter is either a simple one or a data structure, all of whose elements are used by the called class or object. For a good design, classes or objects must have a low *coupling*, but highly coupled superclasses and subclasses are an aid to inheritance, which is very important for object-oriented design.

(iii) ***Sufficiency*** refers to classes capturing enough features of the abstraction to permit meaningful and efficient interaction. For example, if one designs the class *Houses*, one must remember to include an operation which removes an item from the class, but if one should neglect an operation which adds an item, the original idea is a waste.

(iv) ***Completeness*** refers to the interface of the class which should capture all of the meaningful features of the abstraction. Sufficiency implies a

minimal interface. A complete class is thus one whose interface is general enough to be commonly usable to any client. Because completeness may be overdone, it is suggested that classes should be primitive.

(v) ***Primitive*** operations are those which may be efficiently implemented only if given access to the underlying representation of the abstraction. For example, adding an item to a class is primitive, because to implement this *Add* operation, the underlying representation must be visible. On the other hand, an operation adding four items to a class is not primitive since this operation may be implemented just as efficiently upon the more primitive *Add* operation, without having access to the underlying representation.

2.7 Knowledge-based System for Design

A knowledge-based system is a computer-based consultant which has access to stored expertise about some problem domain which is normally performed by a skilled human (Cronk, Callahan & Bernstein, 1988). A knowledge-based system for object-oriented design is a system which is able to assist the system designer to use the expert knowledge of other system designers in order to create a good object-oriented design. It has already been established that it is possible to build a knowledge-based system as an aid for system designers in information system design (Bouzeghoub, 1985). The aim of this research is to build a knowledge-based system by means of a selected knowledge-based environment which will assist the system designer in applying the selected design method as well as in making decisions. For example, where may prototyping be useful during the design process, or where is iteration in the Design Cycle possible?

2.8 Summary and Conclusions

The Software Process Model has been discussed and the Management Aspect, the Life-Cycle Aspect, and the Methods Aspect of the Development Process (DP) Reference Model have been explained.

A multi-perspective view for information systems has been described. It is true that a methodology which concentrates on all three perspectives is the ideal.

Categories of structured design methods have been summarized, namely top-down structured design and data-driven design.

First of all, object-oriented design was discussed by reviewing the principles of object-orientation. Secondly, the object-oriented design process and how it interfaces with object-oriented analysis and object-oriented programming was explained. Thirdly, the object-oriented analysis and object-oriented design methods were summarized. After this literature study, the author decided that the OMT methodology (Rumbaugh *et al.*, 1991) would be used for purposes of this research. The reasons are that it is an object-oriented methodology which supports the whole of the software development life cycle and that it is a methodology which supports a multi-perspective view on any specific problem domain.

Good design principles were explained by referring to high cohesion, low coupling, sufficiency, completeness and primitiveness.

It is concluded that knowledge-based support for object-oriented design is possible because knowledge-based support for conventional structured systems does indeed exist.

CHAPTER 3
Knowledge-based Systems

CONTENTS

3.1 Introduction

In Chapter 2 the design process was discussed. It seems to be a complex task, long and iterative, and full of uncertainty. The nature of the task of design is two-sided. Firstly there is an algorithmic part, for example following a certain methodology, and secondly a heuristic part, for example experimental rules of system designers. When supporting the design process by means of a support system, the support system must be able to include both formal knowledge and experimental knowledge. For all these reasons it seems appropriate to assist the design process by giving advice to system designers by means of a knowledge-based system. This chapter reviews the knowledge-based technology as it pertains to the objectives of the investigation and justifies the choice of Kappa-PC.

3.2 A Knowledge-based System

In a knowledge-based system the problem domain knowledge is explicit and separate from the general knowledge, for example knowledge about how to solve problems. The collection of the domain knowledge is called the *knowledge base*, while the general problem-solving knowledge is called the *inference engine.*

3.3 An Expert System

An expert system is a system which is an "expert" in some narrow problem area. It may ease the work of the expert system user by making available the expert knowledge of others in order to solve complex problems and render advice or recommendations. These systems usually represent knowledge symbolically, their reasoning processes are examined and explained by means of on-line help facilities or on-line queries, and they address problem areas which require years of special training and education for humans to master. Thus, it is possible to provide explanations and the relevant rules used when the system offers particular proposals regarding a problem. Expert Systems are Knowledge-based Systems as explained in *Figure 3.1.*

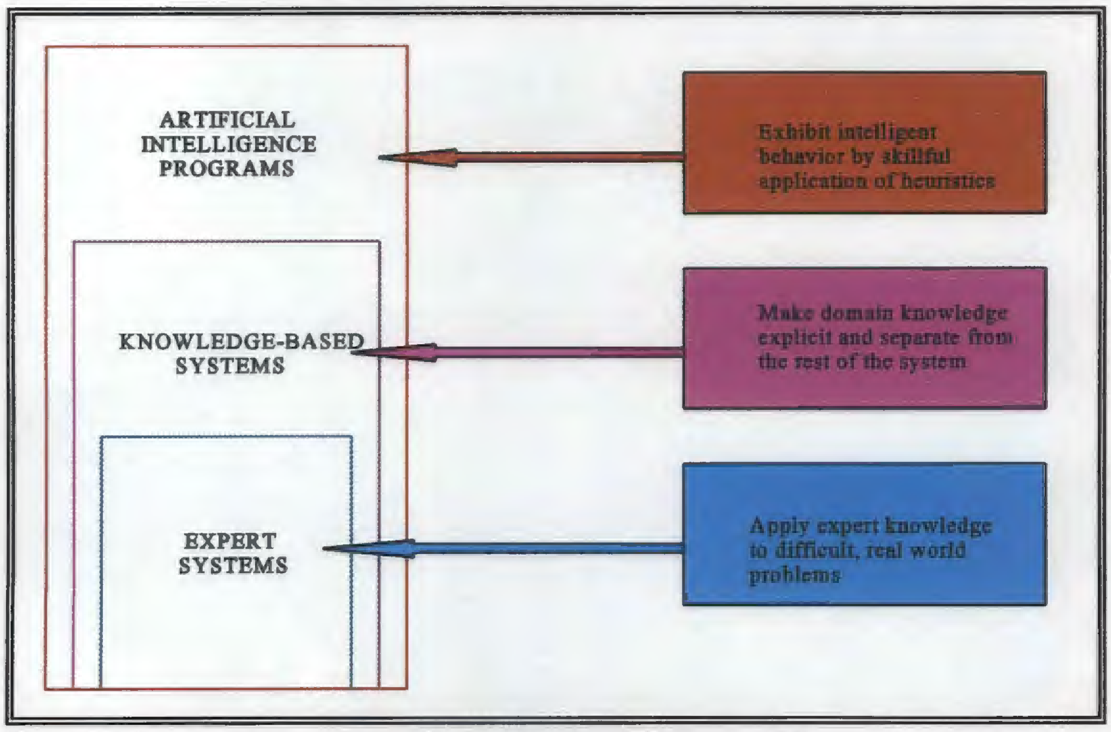

ARTIFICIAL INTELLIGENCE PROGRAMS
KNOWLEDGE-BASED SYSTEMS
EXPERT SYSTEMS
Exhibit intelligent behavior by skillful application of heuristics
Make domain knowledge explicit and separate from the rest of the system
Apply expert knowledge to difficult, real world problems

Figure 3.1 Expert systems are knowledge-based systems (Waterman, 1986)

3.3.1 The Structure of an Expert System

An expert system consists of a user interface, a knowledge-base and an inference engine. *Figure 3.2* is a combination of a figure from Waterman (1986) and a user interface component.

A *User Interface* is a language processor for friendly, problem-oriented communications between the user and the computer. This communication may be in a natural language, extended with menus and graphics (Turban, 1990). A *Knowledge-base* contains lots of detailed knowledge about a particular problem domain. The knowledge may be represented as facts (what is known about the problem area) and rules (logical references between facts) which state *what* the knowledge is. This implies that the knowledge-base is non-procedural.

An *Inference Engine* contains knowledge about *how* to make effective use of the domain knowledge, for example how to solve the problem or how to interact with the user. This implies that the inference engine is highly procedural. It consists of an interpreter and a scheduler.

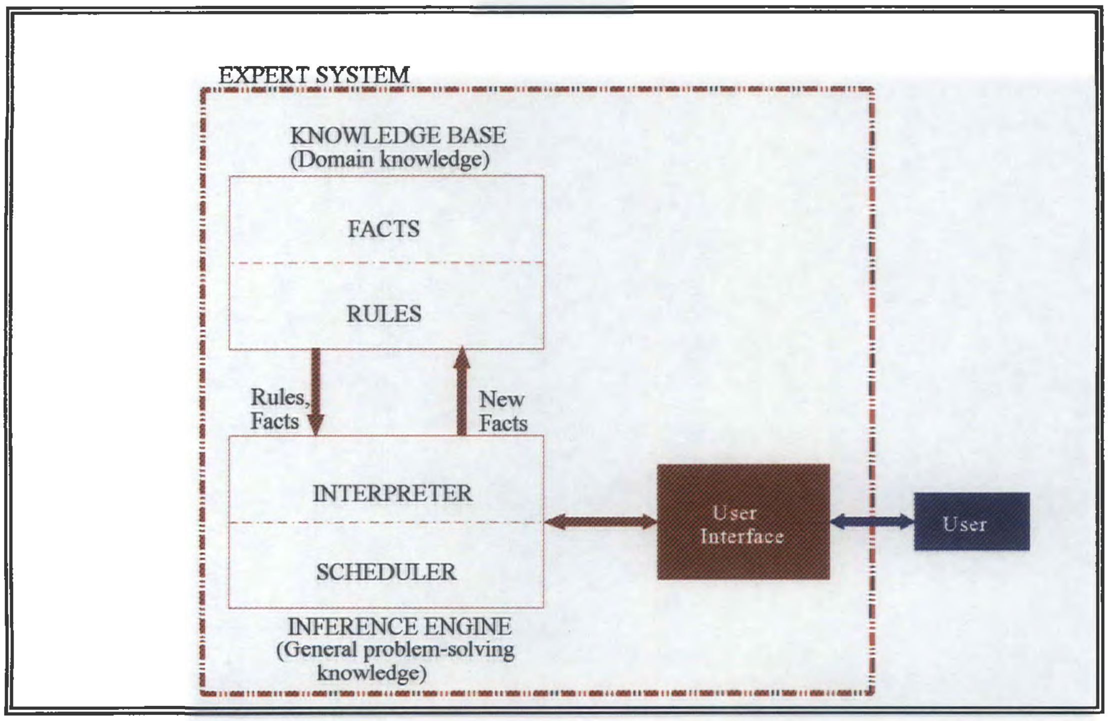


Figure 3.2 The structure of an expert system

3.3.2 The Main Players in an Expert System

When considering expert systems, the main players in this "game" are (Waterman, 1986):

(i) The knowledge engineer,

(ii) the domain expert,

(iii) the end-user and

(iv) the expert system building tool.

Their basic role and relationship to each other is illustrated in *Figure 3.3*.

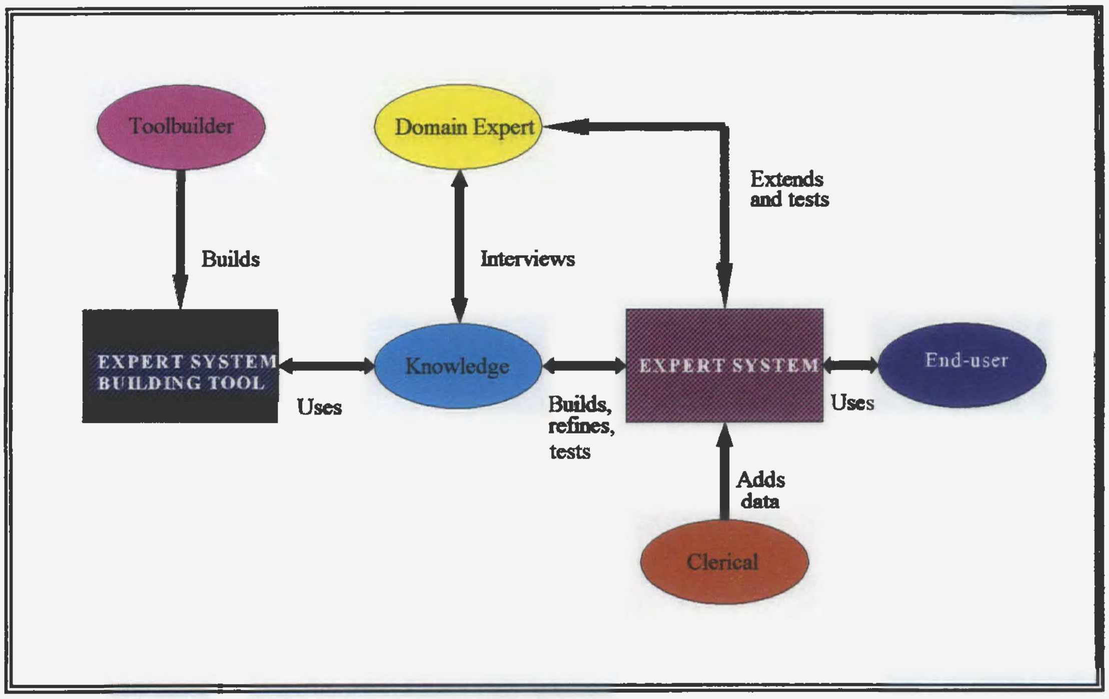


Figure 3.3 The players in the expert system game (Waterman, 1986)

Knowledge-engineering is the process of building an expert system. The expert-system builder, called the *knowledge engineer*, obtains the procedures, strategies and rules of thumb for problem-solving from a human *domain expert*. He then builds this knowledge into the expert system. This expert system will solve problems in much the same way as the domain expert, and the *end-user*, for whom the expert system was developed, will be able to make use of the expert knowledge without the availability of the real domain expert. *Expert system building tools* are available to build expert systems. Forsyth (1989) and Waterman (1986) talk about these expert system building tools and in *Figure 3.4* they are visually depicted. They are:

(i) Programming languages,
(ii) knowledge engineering languages,
(iii) system-building aids and
(iv) support facilities.

(i) ***Programming Languages*** are either ***problem-oriented languages***, such as PASCAL and FORTRAN, or *symbol-manipulation languages*, such as LISP and PROLOG. LISP is especially efficient for work in Artificial Intelligence.

(ii) ***Knowledge Engineering Languages*** consist of an expert system building language integrated into an extended support environment. Knowledge engineering languages are either *skeletal* or *general-purpose*. A skeletal knowledge engineering language is a stripped-down expert system, also called an expert system shell. An expert system shell is an expert system with its domain-specific knowledge removed. The inference engine and support facilities form part of the shell. A general purpose knowledge engineering language may handle different problem areas and types.

(iii) ***System-Building Aids*** consist of *programs which help capture and illustrate* the domain expert's knowledge and *programs which design* the expert system

under construction. Many of these aids are research tools just beginning to mature into functional and effective aids.

(iv) ***Support Facilities*** help with *programming*, for example debugging aids and knowledgebase editors. They also strengthen and *explain* the potential of the finished product, for example built-in input/output facilities and explanation facilities. The Support Facilities are usually combined with a Knowledge Engineering Language and are designed to work specifically with that language.

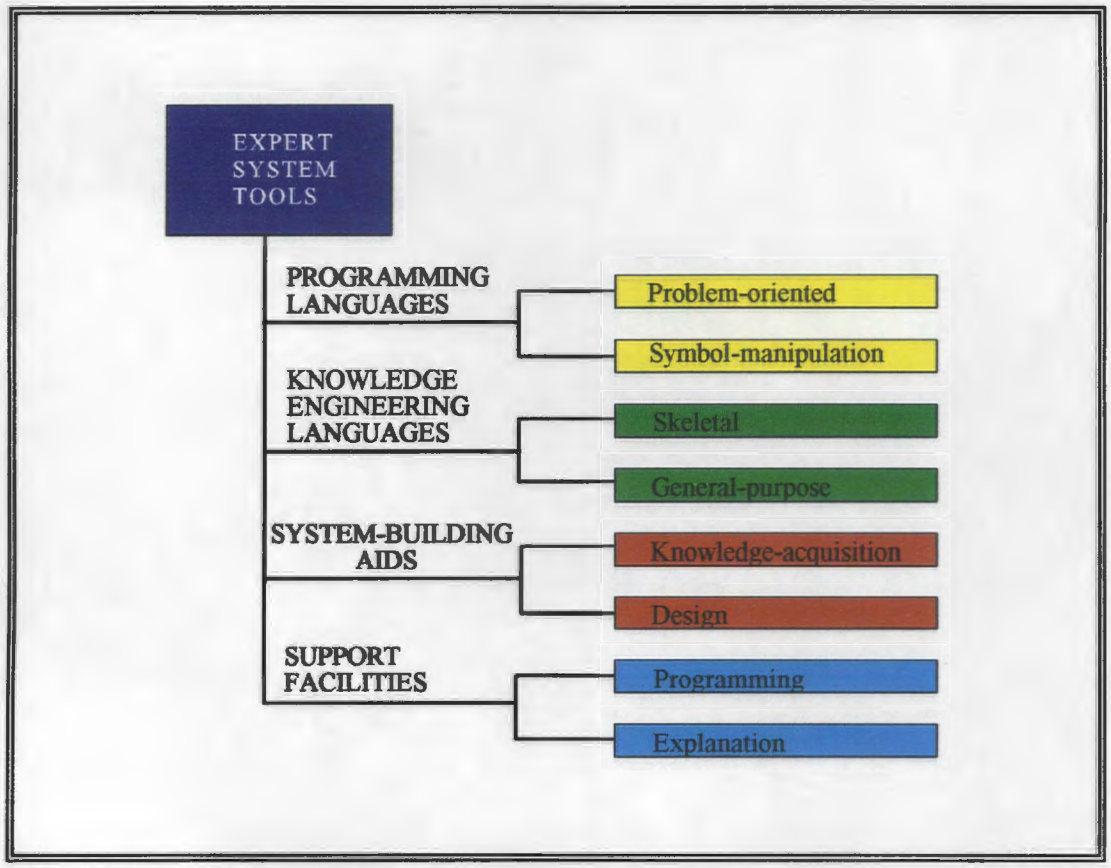


Figure 3.4 Types of tools available for expert system building (Waterman, 1986)

3.3.3 Basic characteristics of an Expert System

The characteristics of an expert system which distinguish it from a conventional program are (Waterman, 1986 and Turban, 1990):

(i) Expertise
(ii) Symbolic Reasoning
(iii) Depth
(iv) Self-knowledge (Explanation Facility).

(i) ***Expertise*** refers to expert systems demonstrating skilful performance, having a high level of competence, and having adequate depth and breadth in a subject.

(ii) ***Symbolic Reasoning*** is the concept in terms of which expert systems represent knowledge symbolically, and manipulate and reformulate symbolic knowledge. Most current expert systems do not have the latter capability.

(iii) ***Depth*** in an expert system means that it operates best in a narrow domain containing challenging problems by using complex rules (meaning complex through their individual complexity or their great numbers).

(iv) ***Self-knowledge (explanation facility)*** refers to an expert system examining its own reasoning and explaining its operation.

3.4 Knowledge Acquisition, Representation and Inferencing

Knowledge is fundamental to the operation of expert systems. The important questions about knowledge are: How does one accumulate knowledge? How does one represent knowledge? and How are conclusions made about this knowledge? Chabris (1988), Turban (1990), and Waterman (1986) have the following to say about these questions:

(i) ***Knowledge acquisition*** is the accumulation, transfer, and transformation of knowledge, derived from various sources, especially from experts, so that it may be symbolically represented and processed. Other potential sources of knowledge include textbooks, databases, special research reports, and pictures. The knowledge engineer must perform this accumulation and reformulation of the knowledge.

(ii) ***Knowledge representation*** is a process of structuring knowledge (facts and rules) about a problem in the computer, in a way which makes the problem easier to solve. The three knowledge-representation schemata that are most commonly used for knowledge representation are rules, semantic nets and frames.

- A *Rule* is a formal way of defining a suggestion, directive, or strategy expressed as

  IF *premise* THEN *conclusion*

  or

  IF *condition* THEN *action.*

  In a rule-based expert system, the domain knowledge is symbolized as sets of rules which are checked against a collection of facts about the current situation. When the IF portion of a rule is satisfied by the facts of the current problem, the action specified by the THEN portion is performed.
- A *Semantic Net* is a representation scheme consisting of a network of

points, called nodes (standing for events, concepts or objects), connected by links, called arcs, describing the relations between the nodes. One of the most common relationships in semantic networks (Turban, 1990) is the ***is a*** link, which allows facts to be attached to classes of objects (for example Poodle ***is a*** Dog), and the ***has a*** link, which allows facts to be inherited by specific objects in the class (for example Dog ***has a*** Tail).

- A *Frame* is a representation scheme which uses a network of nodes (representing concepts or objects) and relations organized in a hierarchy. The concept at each node is defined by a collection of attributes (called slots) and values of those attributes. Each slot may have procedures attached to it which are executed when the information in the slot is changed.

(iii) ***Inferencing*** is the technique used by the inference engine to access and apply the domain knowledge. An inference is a conclusion based on facts or premises. A control mechanism controls the way the reasoning strategy is applied. Examples of control mechanisms are forward-chaining and backward-chaining. Forward-chaining means to chain forward from conditions which are true, towards conclusions which the facts allow one to establish. Backward-chaining refers to chaining backwards from a conclusion one wishes to establish, towards the conditions necessary for its validity, to see if they are supported by the facts.

3.5 Selection Criteria for Expert System Development Environments

The evaluation and selection of a specific expert system environment are important parts of the demonstration of this research. A systematic process (Stylianou *et al.*, 1992) was used for the identification of Kappa-PC, the expert system environment, which was used for this research. The expert system environment evaluation criteria are grouped into eight categories:

(i) End-User Interface Criteria
(ii) Developer Interface Criteria
(iii) System Interface Criteria
(iv) Inference Engine Criteria
(v) Knowledge Base Criteria
(vi) Data Interface Criteria
(vii) Cost-Related Criteria
(viii) Vendor-Related Criteria.

For each category the criteria considered most important are double underlined. The criteria next in importance are underlined.

(i) End-User Interface Criteria

With expert systems the end-user is as important as in any other kind of computer software. Regardless of the specific knowledge captured in an expert system and the development capabilities which the expert system environment offers, if the end-user is not satisfied the project will fail. The following end-user interface criteria are important:

- Saved Cases give the user the opportunity to interrupt his communication with the system, and later be allowed to re-enter and continue from the point of interruption without having to start over.

- Explanation Facilities for expert users will be appreciated, for example:
  - Showing the Reasoning Path with a How Graph
  - Offering Paraphrases to answer "What" questions
  - Answering "Why" questions by pointing Relevances out
- Documentation will facilitate the use of an expert system.
- Tutorial which is good, will make the understanding of the expert system easier.
- Windows are usually very user-friendly. When adding
  - Window Colors, Borders, and Sizes or a
  - Menu System with
    - Pop-Up Menus or
    - Pull-Down Menus, an expert system become easier to use.
  - Customizable Features for end-users where they may design custom screens from a screen design toolkit.
- Speech I/O for voice recognition and/or synthesis.
- Accepts Unknown as an Answer makes the communication for the end-user easier.
- Context-Sensitive Help helps the end-user to help himself.
- Display Manager should offer
  - Graphic Results which are easy to understand, and
  - Graphic Decision Trees which help to trace logic.
- Optimization of displays is important because cluttered and confusing displays encourage user resistance.

- Learning facilities will make it easier for the end-user.
- Mouse Support is important to some end-users.
- Natural Language Interface makes the use of an expert system easier by helping with the communication.
- Sensitivity Analysis and Change Answers and Rerun makes working with, and debugging the system, quicker.

(ii) Developer Interface Criteria

If the developer interface is good and easy to use, the developer will be more productive and efficient. The following developer interface criteria are important:

- Command Language and interpreters are features which facilitate rapid prototyping, which is critical for expert system development.
- <u>Documentation</u> is critical for the developer.
- <u>Tutorials</u> which are good, may not be ignored.
- Editing and Debugging Tools such as
    - <u>Rule and Working-Memory Browsers</u> which allow the developer to view every link between rules,
    - <u>Tracing</u> for observing the chain of events,
    - <u>Cross-Index Utility</u> for, amongst others things, the creation of a back-up when the code is modified, and
    - <u>Incremental Compilation</u> are considered to be very important because they speed up the development process.
- <u>Explanation Facilities</u> such as
    - How certain conclusions were made (reasoning path)

  - What is the meaning of the question being asked (paraphrase) and
  - Why is that question being asked (relevance), gain the developer's confidence and are very good debugging tools.
- Ability to Customize Explanations makes the system more understandable.
- Graphics always enhances clarity.
- Mathematical Capabilities add an important feature in many applications.
- Sample Knowledge Bases may minimize the developer's work.
- Code Generator may ease or eliminate many programming problems.
- Windows are usually very user-friendly. When adding
  - Window Colors, Borders, and Sizes or a
  - Menu System with
    - Pop-Up Menus or
    - Pull-Down Menus, an expert system become easier to use.
  - Customizable Features where a developer may design custom screens from a screen design toolkit.
- Rapid Prototyping is very important for demonstration purposes, for example when needing management acceptance.
- Open Architecture enhances the portability of the system.
- Batch-Processing Facilities are a help for the developer.
- Novice and Expert Modes will help not to frustrate a novice or expert developer.
- String Handling where steps may be combined by using shortcuts and command macros will be very useful to the developer.

(iii) System Interface Criteria

The system interface criteria concentrate on the available hardware, certain features of the implementation language, copy protection, batch processing, real-time processing, and network support.

- The Hardware spectrum is very important in the sense of servicing a big audience of end-users.
  - Portability makes development on one machine and usage on another machine possible.
  - Support for Microcomputers introduces a broad spectrum of end-users to such an expert system.
  - Compatibility with standard computer environments is important.
  - Multi-processor Support and
  - Multi-user Support also broadens the end-user spectrum.
  - Access to Special Hardware is very convenient.
- Implementation Language must be powerful in the sense of
  - Portability, which means expert systems must operate efficiently within mainstream computer environments,
  - Embeddability refers to the ability of expert systems to be built into conventional applications, thereby providing these applications with the advantages of a knowledge-based system, and
  - Compatibility, when a newly developed expert system will operate within the existing systems environment.
- Copy Protection entails protecting the source code before handing the

system to the end-users, and the capability of preventing the end-user from accessing and damaging the knowledge base. Passwords, encryption and read/write privileges are important.

- Batch Processing,
- Real-Time Processing and
- Network Support make working with a shell more productive.

(iv) Inference Engine Criteria

The inference engine is a collection of programming routines which implement one or more reasoning modes, search techniques, conflict resolution strategies, uncertainty handling systems, tracing and error checking.

- Reasoning Mode consists of:
  - Forward Chaining (data-driven), where the system begins with known facts, trying to assert new facts.
  - Backward Chaining (goal-driven), means the system starts with a goal or hypothesis and tries to match that goal with the action clauses.
  - Bi-Directional Inferencing combines forward and backward reasoning.
  - Non-monotonic Reasoning is the process whereby facts may be changed after they have been established. These systems may deal with very dynamic problems involving rapid changes in values in short periods of time.
- Truth Maintenance System is a way of keeping track of postulates and their justifications developed during an inferencing process.

- Search Strategy takes various forms:
  - Breadth First, when every item at a given level is evaluated before proceeding to the next level.
  - Depth First concentrates on evaluating only one item at a given level before proceeding to the next level.
  - Branch-And-Bound means generating complete reasoning paths and keeping track of the shortest path found so far.
  - Generate-And-Test
  - Best First refers to moving forward from the node which seems closest to the goal node.
  - Hill Climbing is depth-first with a heuristic measure which orders choices when branching points are reached.
- Find All Answers and
- Find Only One Answer are both important and necessary under specific conditions.
- Conflict Resolution decides which rule should be activated whenever there is a conflict, for example:
  - Rule-Assigned Priority gives the developer complete control.
  - Specificity points out exactly which rule should be applied.
  - Recency chooses the rule because of the collection of facts which have been established more recently than the facts used by the other rules.
- Certainty Measurement is a method of dealing with uncertain or incomplete user input and imprecise knowledge. The different paradigms are:
  - Bayes Theorem (Forsyth, 1989), which rests on the belief that

for everything, no matter how unlikely it is, there is a prior probability that it could be true. It may be a very low probability, in fact it may be zero, but it does not prevent us from calculating as if there were a probability there.

- Certainty Factor Model (or MYCIN model)(Chabris, 1988) (Waterman, 1986), where a certainty factor is associated with each piece of data in a working memory (in the MYCIN expert system) and with each conclusion it draws in its reasoning process. The value of certainty factors ranges from -1.0, representing absolute untruth of a proposal, to 1.0, representing absolute truth or confidence. These certainty factors are chosen arbitrarily by the expert himself.
- Dempster-Shafer Theory (Lucas & Van der Gaag, 1991), where current evidence leads to multiple beliefs regarding the same hypothesis. This theory combines the beliefs in order to compute an overall measure of belief in the hypothesis.
- Fuzzy Set Theory (Ford, 1991) (Shinghal, 1992) (Turban, 1990) is suitable for solving problems which involve entities defined by vague terms such as "about".
- <u>Inheritance</u> refers to facts or rules previously tested to be valid.
- Certainty Threshold considers only outcomes which have more than a certain percentage of certainty.

- Blackboard, where data may temporarily be stored.
- <u>Recursion</u> and
- <u>Iteration</u> in the inference engine make the engine more powerful.
- Fuzzy Sets enables the inference engine to react less precisely and

logically than usual.

- Reliability in the inference engine is very important.

(v) Knowledge Base Criteria

Knowledge base criteria concentrate on the knowledge engineering sub-system, representation technique for the knowledge, inheritance, multiple instances, ability to generate a decision tree and/or a set of rules demonstrating the expert's decision process, and a few more which will also be discussed.

- Representation Techniques may be one, or a combination of the following:
  - Rules, in the form of a series of production rules, represent the knowledge.
  - Partitioned Rule Sets, where rules are partitioned according to some criteria - for example, where all validation rules are grouped in one set, and all verification rules are grouped in another set.
  - Meta-rules are rules which contain knowledge on how to process standard rules. They provide an index to the rest of the knowledge base.
  - Decision Tables refer to structuring the knowledge in the form of tables.
  - Frames allow objects to be associated with collections of features. Each feature is stored in a slot. Each frame is composed of a set of slots related to a specific object.
  - Scripts (or Schemata) represent knowledge regarding

accumulated events, taking place in familiar situations, in a series of "slots". A script is composed of a series of scenes which are, in turn, composed of a series of events.

  - o Semantic Networks represent objects as nodes which are connected to other nodes by arcs. These networks represent the relationship among objects.
  - o Formal Logic refers to "recasting" various knowledge representations in terms of logic. This leads to a better understanding of knowledge representation and logic which may handle incompleteness and default reasoning (Turban, 1990).

- Induction is the capability of an environment to generate a decision tree and/or a set of rules from a set of examples demonstrating the expert's decision process.
- <u>Inheritance</u>, where one object inherits properties of other objects higher up in the hierarchy. Inheritance may eliminate duplication and redundancy in knowledge representation.
- <u>Knowledge Engineering Sub-system</u>, which orchestrates the following activities: Knowledge acquisition, knowledge representation, inferencing, and explanations and justifications (Turban, 1990).
- <u>Multiple Instance</u>, where two or more knowledge representation schemata are used.
- Demons are procedures which are automatically activated by the changing or accessing of values in the knowledge base (Turban, 1990).
- Case Management organizes case information, estimates case value, and suggests tactics and strategies for negotiation and case settlement in expert systems for law (Waterman, 1986).

- Capacity of the knowledge base is important.

(vi) Data Interface Criteria

Developers of expert systems often find it necessary to cross the boundaries of the shell environment. A capability might be needed which is best implemented somewhere else or which is not provided by the shell. For this reason the following features are important:

- Access to 3GL and 4GL
- <u>Linkage to Databases</u>
- <u>Access to Underlying Language</u>
- <u>Linkage to Special Purpose Software</u>
  - Linkage to Transaction Processing Environments
  - Access to Lotus, DOS, etc.

(vii) Cost-Related Criteria

"The pricing of tools is confusing. Some less powerful products are priced high, while some of the cheaper products are very credible." (Harmon, Maus & Morrissey, 1988). The following are important cost-related criteria, but not important enough, according to the relevant article, to be underlined:

- Upgrades
- Required Software/Hardware
- Conversion
- Personnel
- Vendor Technical Support
- Training Programs

- Installation
- Run-Time Licence
- Consulting Fees.

(viii) Vendor-Related Criteria

According to Holsapple and Whinston (1987), vendors with a continuing history of introducing software modernization are more likely to offer a shell which is close to the state of the art. Another positive indication is the vendor's track record of enduring enhancements of their software products. The following criteria are considered to be important.

- Maintenance
- Technical Support
- Training Courses
- Professional Application Development Services
- Product/Vendor Maturity
- Commitment to Product
- Upgrade Path.

3.6 Evaluation of Candidate Expert System Development Environment

A number of environments were evaluated, namely Kappa-PC, [1]Leonardo, [2]Nexpert Object, [3]ART-IM, and [4]EXSYS Professional. The evaluation was of

[1] Leonardo is a registered trademark of Creative Logic.

[2] Nexpert Object is a registered trademark of Neuron Data.

[3] ART-IM is a registered trademark of Inference Corporation.

[4] EXSYS Professional is a registered trademark of Exsys, Inc.

necessity, based on literature and demonstrations of the vendors. A systematic process (Stylianou *et al.*, 1992) was followed, without hands-on experience of the environments. The results are summarized in *Appendices A, B, C, D* and *E* respectively and should be seen in the context of the emphasis placed on particular criteria by the evaluation process followed. The Kappa-PC system was chosen based on the summarized rating of 56. Other scores were Leonardo (53), Nexpert Object (15), ART-IM (28), and EXSYS Professional (26). After personally working with Kappa-PC the evaluation for Kappa-PC was extended, for the purposes of this investigation. In this section the revised results, according to the author, are presented.

(i) End-User Interface Criteria

• Saved Cases	Indirect	Must program capability
• Explanation Facilities		
○ Reasoning Path - How Graph	YES	
○ What - Paraphrases	Indirect	Program own explanation facility and directly reference from rules or monitors
○ Why - Relevances	Indirect	As above - plus use meta-rules to establish relevant rule sets
• Documentation	YES	
• Tutorial	YES	
• Windows		
○ Window Colors, Borders, Sizes	YES	
○ Menu System		
▫ Pop-Up Menus	YES	
▫ Pull-Down Menus	YES	
○ Customizable Features	YES	
• Speech I/O	Indirect	Using a 3rd party product e.g. Dynamic Link Library (DLL)
• Accepts Unknown as an Answer	YES	All AskValue and

		PostInputForms allow user to enter Unknown i.e. NULL
● Context-Sensitive Help	Indirect	Using Windows own help system and a simple DLL call
● Display Manager		
○ Graphic Results	YES	
○ Graphic Decision Tree	YES	
● Optimization	YES	
● Learning	YES	Help systems for the user
● Mouse Support	YES	
● Natural Language Interface	Indirect	Using a 3rd party product e.g. DLL interface
● Sensitivity Analysis or Change Answers and Rerun	YES	The inference engine may be rerun without resetting all user input values except those one wishes to change

(ii) ***Developer Interface Criteria***

● Command Language/interpreter	YES	
● Documentation	YES	
● Tutorial	YES	
● Editing/Debugging Tools		
○ Rule/Working-Memory Browser	YES	
○ Tracing	YES	
○ Cross-Index Utility	NO	
○ Incremental Compilation	NO	But it may compile any part of the application to C at any time and re-integrate as a DLL
● Explanation Facility		
○ How (Reasoning Path)	YES	
○ What (Paraphrase)	Indirect	See previous comment

○ Why (Relevance)	Indirect	See previous comment
● Ability to Customize Explanations	YES	It is relatively straightforward to code one's own customised explanation facility - as above
● Graphics	YES	
● Mathematical Capabilities	YES	The KAL language is a comprehensivegeneral-purpose language supporting a wide range of maths functions
● Sample Knowledge Bases	YES	
● Code Generator	YES	Generates C code from KAL
● Windows		
○ Window Colors, Borders, Sizes	YES	
○ Menu System		
▫ Pop-Up Menus	YES	
▫ Pull-Down Menus	YES	
○ Customizable Features	YES	
● Rapid Prototyping	YES	Kappa-PC is an ideal tool for rapid prototyping. It supports all the necessary elements e.g. rich graphical tools, interpreter, dynamic object engine, GUI builder, etc.
● Open Architecture	YES	Kappa-PC supports a C Application Programming Interface (API), DLL, Dynamic Data Exchange (DDE), and SQL interface as well as generating C. It is extremely open and easy to integrate and embeds with other applications
● Batch Processing Facilities	YES	

● Novice/Expert Modes	NO	
● String Handling	YES	KAL supports full string manipulation e.g. SubString, FindSubString,StringLength, #, etc.
(iii) System Interface Criteria		
● Hardware		
○ Portability	Limited	
○ Support for Microcomputers	YES	
○ Compatibility	YES	Kappa-PC is fully compatible with other applications running under MS-Windows in terms of look-and-feel and the DDE and DLL interfaces
○ Multi-processor Support	NO	
○ Multi-user Support	NO	
○ Access to Special Hardware	Indirect	Any real-time hardware card or specialised control card may be accessed via the C API
● Implementation Language		
○ Portability	NO	
○ Embeddability	YES	Once compiled to C or directly via DDE from another application
○ Compatibility	YES	Kappa-PC is fully compatible with other applications running under MS-Windows in terms of look-and-feel and the DDE and DLL interfaces
● Copy Protection	YES	Source code and knowledge bases may be completely

		protected by compiling them into C - password capability is also possible on all Kappa-PC edit boxes
• Batch Processing	YES	
• Real-Time Processing	YES	
• Network Support	YES	
(iv) Inference Engine Criteria		
• Reasoning Mode		
o Forward Chaining	YES	
o Backward Chaining	YES	
o Bi-Directional Inferencing	YES	
o Non-monotonic Reasoning	Indirect	An appropriate algorithm would need to be coded in KAL
• Truth Maintenance System	Indirect	A Truth Maintenance System (TMS) could be coded in KAL but its performance would not be optimized, which is critical for TMSs
• Search Strategy		
o Breadth First	YES	
o Depth First	YES	
o Branch-And-Bound	Indirect	Can be coded in KAL
o Generate And Test	Indirect	Can be coded in KAL
o Best First	YES	
o Hill Climbing	Indirect	Can be coded in KAL
• Find All Answers	YES	
• Find Only One Answer	YES	
• Conflict Resolution	YES	
o Rule-Assigned Priority	YES	

○ Specificity	YES	The "SELECTIVE" strategy
○ Recency	Indirect	Recency would need to be added to the domain object model and accessed from within rules
● Certainty Measurement		Generally Kappa-PC does not directly support any particular methods for uncertainty handling - instead the emphasis is on the developer to code KAL functions to combine uncertainties or possibilities according to some given algorithm, e.g. Bayes, and to call these directly from within rules and methods
○ Bayes Theorem	Indirect	See above note
○ <u>Certainty Factor Model</u>	Indirect	See above note
○ Dempster-Shafer Theory	Indirect	See above note
○ Fuzzy Set Theory	Indirect	See above note
○ <u>Inheritance</u>	Indirect	See above note
○ Certainty Threshold	Indirect	See above note
● Blackboard	Indirect	The domain object model may be viewed as a blackboard and event monitors linked to these objects may trigger particular rule sets or methods to change the state of the "blackboard"
● <u>Recursion</u>	YES	KAL is a fully recursive language
● <u>Iteration</u>	YES	KAL supports full iteration e.g. For x From 1 To 10 Do

• Fuzzy Sets	Indirect	Fuzzy Sets may be programmed in KAL as object classes
• Reliability	YES	
(v) Knowledge Base Criteria		
• Representation Technique		
○ Rules	YES	
○ Partitioned Rule Sets	YES	
○ Meta-rules	YES	It is possible to define rules which control the inference strategy e.g. by changing rule sets, priorities, etc. - these are by definition meta-rules
○ Decision Tables	Indirect	It is possible to code a decision table object in KAL
○ Frames	YES	The objects in KAL are based on Frames - this is where the term "slot" originates from
○ Scripts/Schemata	Indirect	These may be coded as object classes with their appropriate behaviors
○ Semantic Networks	Indirect	May be coded as an object network using KAL
○ Formal Logic	NO	
• Induction	Indirect	Algorithms such as ID3 may be coded using KAL
• Inheritance	YES	
• Knowledge Engineering Sub-system	NO	
• Multiple Instance	Indirect	"Views" may be implemented using KAL
• Demons	YES	KAL's "monitors" are demons
• Case Management	NO	

	• Capacity	Large	Kappa-PC supports up to 500,000 objects and rules
(vi)	***Data Interface Criteria***		
	• Access to 3GL and 4GL	YES	C is a 3GL and Kappa supports a C API
	• <u>Linkage to Databases</u>	YES	
	• <u>Access to Underlying Language</u>	YES	
	• <u>Linkage to Special Purpose Software</u>		
	○ Linkage to Transaction Processing Environments	YES	Using Kappa-PC CommManager or other suitable 3rd party software
	○ Access to Lotus, DOS, etc.	YES	
(vii)	***Cost-Related Criteria***		
	• Upgrades		Maintenance and upgrades for one year: 15% of purchase price
	• Required Software/Hardware		The minimum needed for MS-Windows i.e. 386 with at least 2Mb RAM and 5Mb spare disk capacity
	• Conversion		Depends on existing skill base and products
	• Personnel		Depends on needs
	• Vendor Technical Support		Depends on needs
	• Training Programs		Available and customizable
	• Installation		Depends on needs
	• Run-Time Licence	YES	Highly volume-dependent
	• Consulting Fees		Depends on needs

(viii) Vendor-Related Criteria

• Maintenance	YES	
• Technical Support	YES	
• Training Courses	YES	
• Professional Application Development Services	YES	
• Product/Vendor Maturity	YES	
• Commitment to Product	YES	
• Upgrade Path	Limited	To Kappa (Unix, Windows NT) and OMW (Object Management Workbench)

Kappa-PC was found to be a truly object-oriented development environment. Its interactive, graphical development environment was a real pleasure to work with and its high-level application development language is very powerful. The Kappa-PC expert system tools were of great importance for this investigation. They were used extensively during the demonstration of concept and found to be sound and forceful.

3.7 Summary and Conclusions

Expert Systems commenced from work in Artificial Intelligence laboratories. They are considered to be one of the most successful branches of Artificial Intelligence. Expert systems contain a high density of problem-solving knowledge in a particular application domain. This knowledge allows expert systems to "perform" like the human expert from which the knowledge was acquired.

In a knowledge-based system the knowledge about the problem domain is separated from the general knowledge as to how to solve the problem or how to interact with the user.

The structure of an expert system consists of a user interface, a knowledge base and an inference engine. The main players in an expert system are the knowledge engineer, the domain expert, the end-user and an expert system building tool. The basic characteristics of an expert system are expertise, symbolic reasoning, depth and self-knowledge.

Knowledge acquisition is the accumulation and transformation of knowledge. The knowledge may be represented using representation schemata like frames, semantic nets and rule sets. Accessing and applying the domain knowledge is called "inferencing".

The selection criteria for an expert system environment consist of end-user interface criteria, developer interface criteria, system interface criteria, inference engine criteria, knowledge base criteria, data interface criteria, cost-related criteria, and vendor-related criteria.

Kappa-PC was evaluated against these criteria and found to be the appropriate environment for this research since its development method is truly object-oriented, and it uses extensive rule-based reasoning.

The conclusion that was made after discussing expert systems in general and Kappa-PC in particular is that expert knowledge about object-oriented design, captured by means of an expert system, may be applied when developing a system. By supporting the Design Cycle in such a way, it is possible to guarantee a design of high quality. The expert system guides the following of a sound design methodology, the "pitfalls" are monitored by an "expert", and validation and verification are assisted by an "expert".

CHAPTER 4
OMT - An Object-oriented Design Methodology

CONTENTS

4.1 Introduction

In Chapter 3, knowledge-based systems were addressed because knowledge-based support for the design process is the point of departure. This support must be within the framework of a thorough and trustworthy object-oriented methodology. The methodology which was chosen for this research is Rumbaugh's Object-Modeling Technique (OMT) (Rumbaugh *et al.*, 1991). In Chapter 4 the modeling perspectives of the OMT are reviewed and the OMT method for design is described. The organization of a design knowledge base is proposed, followed by the conceptual model of a proposed solution of the problem under investigation.

4.2 Modeling

The understanding of the requirements of a real-world problem, before building a solution for it, is crucial for the effectiveness and efficiency of the solution. Building a *model* of the function of a proposed solution to a real-time problem, makes the understanding and explanation of that solution easier. A model is an *abstraction* of the presented system for the purpose of understanding it before building it (Rumbaugh *et al.*, 1991). Abstraction enables a person to deal with complexity because it captures those points which are important for some purpose and suppresses those points which are unimportant. The developer must abstract different views of the system, build models according to these views, verify that the models satisfy the user requirements and, step-by-step, add technicalities to transform the models into an implementation.

Models of information systems may be conceptualised in terms of various levels of abstraction (Du Plessis, 1986; Pocock, 1991; Klint, 1993; Harmsen & Brinkkemper, 1993). Each lower level is an instance of the level above it giving definition to the model primitives on each level. The meta model is a high level of abstraction and denotes modeling of a model-object in terms of primary notions called meta primitives. The model primitives at the meta model level are the primary notions required to construct a conceptual model of a real-world problem domain by means of a particular methodology and development environment (Du Plessis, 1994).

4.3 The Object-Modeling Technique

The OMT methodology models a system from three different, but related, viewpoints. The *object model* represents the static and structural perspectives of a system (the "data" perspective). The *dynamic model* represents the temporal and behavioral perspectives of a system (the "control" perspective). The *functional model* represents the transformational perspectives of a system (the "function" perspective). All three perspectives are incorporated in a typical software procedure, for example a software procedure uses data structures (object model), it sequences operations in time (dynamic model) and it transforms values (functional model). Each model contains references to entities in other models. For example, events (dynamic model) become operations on objects (object model), but are more fully expanded as functions in the functional model. The functional model specifies what happens, the dynamic model specifies when it happens and the object model specifies what it happens to.

The meta models, with their primitives (as defined by Rumbaugh, *et al.* (1991)), for the object model, the dynamic model and the functional model are represented in *Exhibits 4.1, 4.2* and *4.3* respectively.

4.3.1 Object Model

The *structure* of objects in a system is described by the object model. The structure of objects encompasses their identity, their relationship to other objects, their attributes and their operations. The construction of an object model has, as its goal, the capturing of those concepts from the real world which are important to an application. An *Object diagram* is the graphical representation scheme of the object model. Object diagrams contain a number of *object classes*. The abbreviation *class* will be used instead of *object class*. Classes are arranged into hierarchies which share common structure and behavior. A class describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects and common semantics. An object is an instance of a class. For example, *Person* is a class. The person Johan Palmer is an *object* (an instance) of the class. An *attribute* is the data value held by the objects in a class. *Name* and *age* are attributes. Each attribute has a value for each object instance. For example, attribute *age* has the value *30* in object *Johan Palmer* and the person Johan Palmer is an object, whose *name* attribute has the value 'Johan Palmer' (the string). *Figure 4.1* illustrates classes, objects and attributes.

An *operation* is a function or transformation which may be applied to or by objects in a class. *Draw* is an operation on class *Circle*. All objects in a class share the same operations. The same operation may apply to different classes, for example the operation *Draw* may also apply to class *Triangle*. This is called "polymorphism", where the same operation behaves differently on different classes. A *method* is the specific implementation of an operation for a class. A

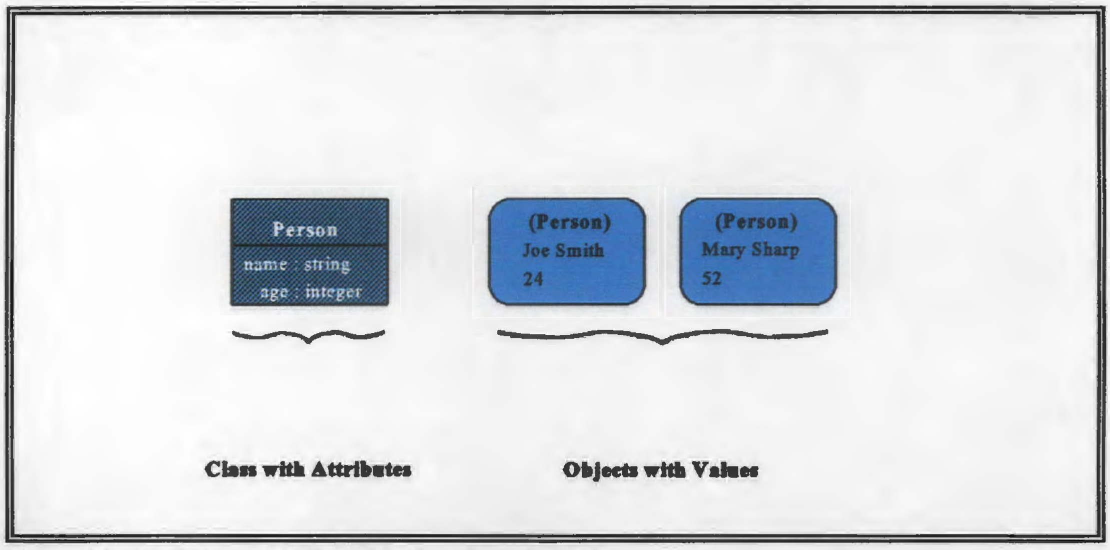


Figure 4.1 Attributes and Values (Rumbaugh *et al.*, 1991)

different method is implemented to *Draw* a figure from class *Circle* than from class *Triangle*. A *link* is a physical or conceptual (theoretical) connection between object instances. For example, Johan Palmer *Lives-in* Pretoria city. A link is an instance of an association. An *association* describes a group of links with common structure and common semantics. *Figure 4.2* illustrates links and associations.

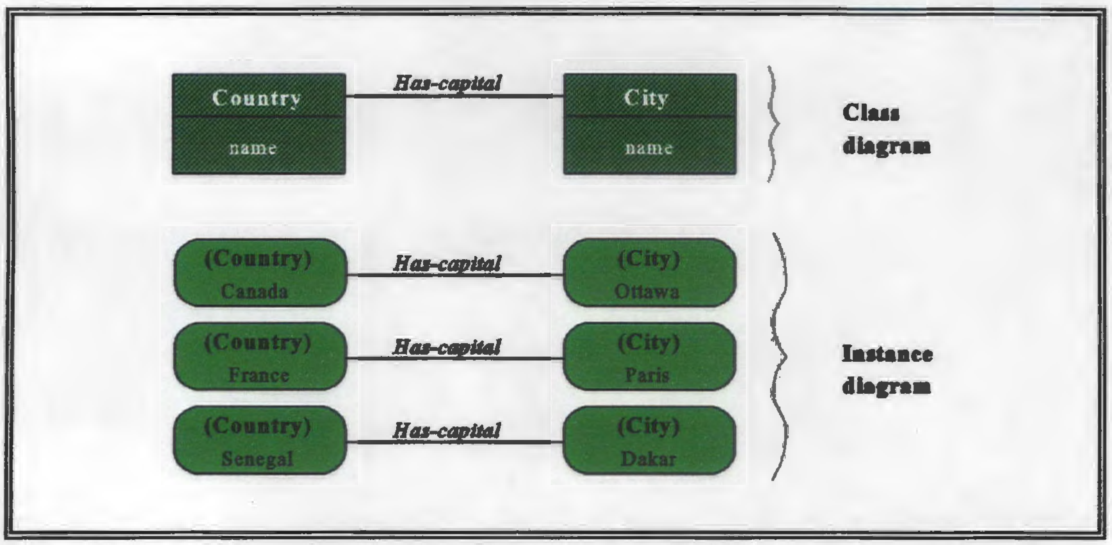


Figure 4.2 One-to-one association and links (Rumbaugh *et al.*, 1991)

The modeling primitives for the object model are presented in *Table 4.1.*

OBJECT	
CLASS	
RELATIONSHIP	aggregation
	association
	association (derived)
	association (qualified)
	association (ternary)
	generalization
	instantiation
	link (between objects)
ATTRIBUTE	antisymmetric (aggregation)
	constraint (association) (class) (link) (object)
	degree (generalization)(aggregation) (association)(instantiation)(link)
	derived attributes (class) (object)
	discriminator (generalization)
	homomorphism (association)
	link (association)
	multiplicity (association)
	ordering (association)
	qualifier (association)
	role (association)
	transitivity (aggregation)
OPERATIONS	abstract operations (class)
	link operations (association relationship)
	operations (class) (object)
	propagation/triggering (class) (object)

Table 4.1 Modeling primitives for the Object Model

These modeling primitives are used to compile the Meta Object Model of *Exhibit 4.1*.

4.3.2 Dynamic Model

The *dynamic model* describes those perspectives of a system concerned with time and the sequencing of operations. The dynamic model captures *control*. Control is the perspective of a system which describes the sequences of operations which occur, without regard for what the operations do, what they operate on, or how they are implemented. A *State Diagram* is the graphical representation scheme of the dynamic model. Each state diagram shows the *state* and *event* sequences permitted in a system for one class of objects. The values of the attributes and links of an object at a particular time, are called its *state*. For example, the state of the engine of a car is either active or inactive, depending on whether its ignition has been switched on or not. The interval between two events received by an object corresponds to a state. An individual stimulus from one object to another is an *event*. An event is something which happens at a point in time, such as *Flight SA23 departs from Jan Smuts*. An event has no duration. The state of the object receiving an event will determine the response to an event. It may include a change of state or the sending of another event to the original sender or to a third object. The pattern of events, states and state transitions for a given class may be abstracted and represented as a state diagram. The dynamic model contains multiple state diagrams, one state diagram for each class with important dynamic behavior. The dynamic model shows the pattern of activity for an entire system. Actions in the state diagrams correspond to functions from the functional model.

The dynamic model specifies allowable sequences of changes to objects from the object model. States are equivalence classes of attribute and link values for the object. Events in a state diagram become operations on objects in the object model. *Figure 4.3* illustrates a state diagram for a phone line.

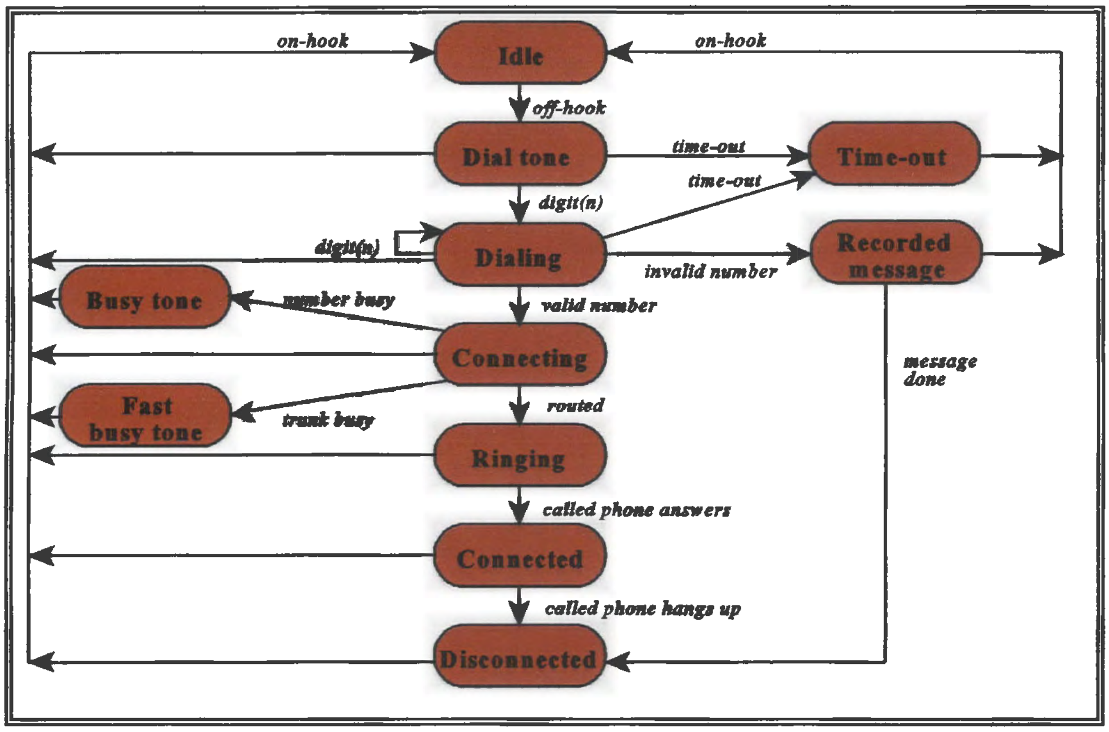


Figure 4.3 State diagram for phone line (Rumbaugh *et al.*, 1991)

The modeling primitives for the dynamic model may be summarized in the form shown in *Table 4.2*.

OBJECT	
CLASS	
STATE	
EVENT	
ACTION	
ATTRIBUTE	event attribute (event)
	final state (state)
	initial state (state)
	intermediate state (state)
	substate (state)
	superstate (state)
	transition type (transition)
RELATIONSHIP	aggregation (state)
	generalisation (event) (state)
	guarded transition
	transition association
CONDITION	guarded condition

Table 4.2 Modeling primitives for the Dynamic Model

These modeling primitives are used to compile the Meta Dynamic Model of *Exhibit 4.2.*

4.3.3 Functional Model

The *functional model* describes those perspectives of a system concerned with transformations of values (computations within a system). The functional model captures what a system does, without considering how or when it is done. The graphical notation for the functional model is the *data flow diagram (DFD)*. Data flow diagrams show the flow of values from external inputs, through operations and internal data stores, to external outputs. A data flow diagram contains *processes* which transform data, *data flows* which move data, *actor* objects which produce and consume data, and *data store* objects which store data passively. *Figure 4.4* shows a data flow diagram for the display of an icon on a windowing system (Rumbaugh *et al.*, 1991). The *icon name* and *location* are inputs to the diagram from an unspecified source. The icon is expanded into vectors, using the icon definition from the *Icon definitions* data store. The vectors are clipped to the size of the window and the location of the window on the screen gives the vector an offset to be able to obtain vectors in the screen coordinate system. Next the vectors are converted to pixel operations and sent to the screen buffer for display. The sequence of transformations performed is shown by the data flow diagram, as well as the external values and objects which affect the computation.

Each *process* in the functional model is implemented by a *method* on some object in the object model. *Actors* in the functional model are explicit *objects* in the object model. *Data stores* in the functional model are also *objects* in the object model. *Data flows* in the functional model are *values* in the object model.

A *process* in the functional model is invoked as an *action* in the dynamic model. The dynamic model for an *actor* object specifies when it acts. *Data stores* are passive objects which respond to queries and updates, and *data flows* are values and pure values have no state and no dynamic model.

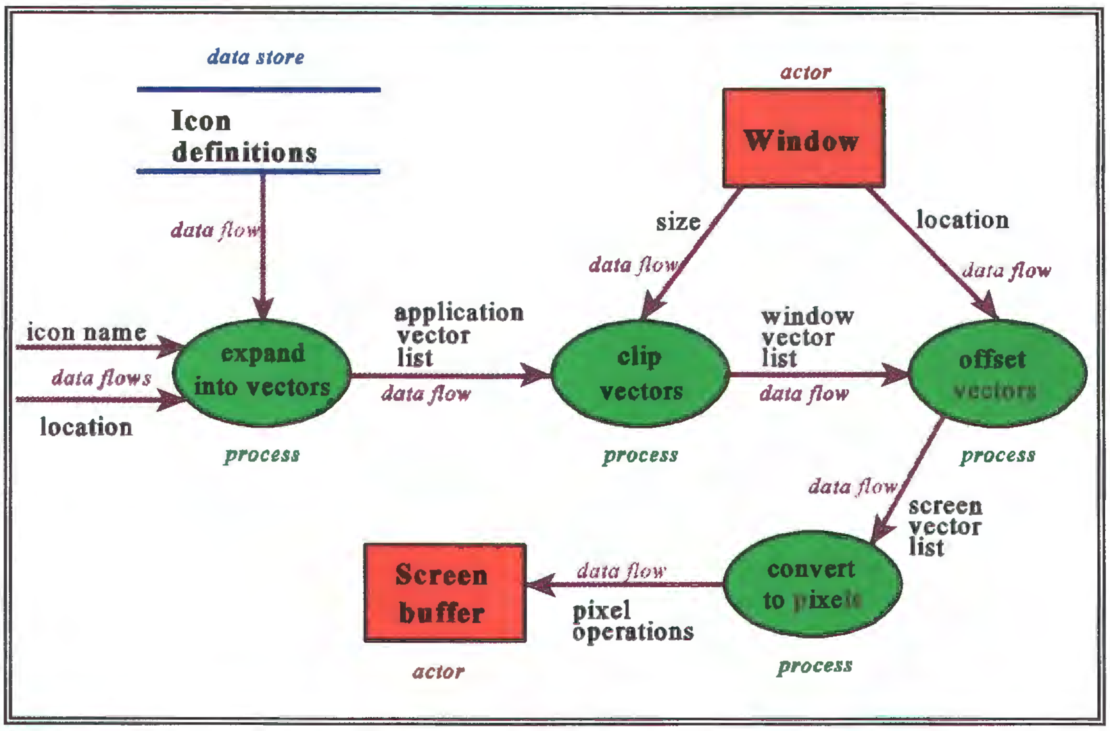


Figure 4.4 Data flow diagram for windowed graphics display (Rumbaugh *et al.*, 1991)

The modeling primitives for the functional model are depicted in *Table 4.3*.

PROCESS	transforms data values
DATA STORE/FILE OBJECT	passive object within DFD
ACTOR OBJECTS	active object that drives data flow graph (sources or sinks of data)
DATA FLOW	connects output of object/process to input of object/process
CONTROL FLOW	Boolean value that determines whether or not a process is evaluated

Table 4.3 Modeling primitives for the Functional Model

These modeling primitives are used to compile the Meta Functional Model of *Exhibit 4.3*.

4.4 Model Summary

The relationships between the modeling primitives of the three models are shown in *Figure 4.5*.

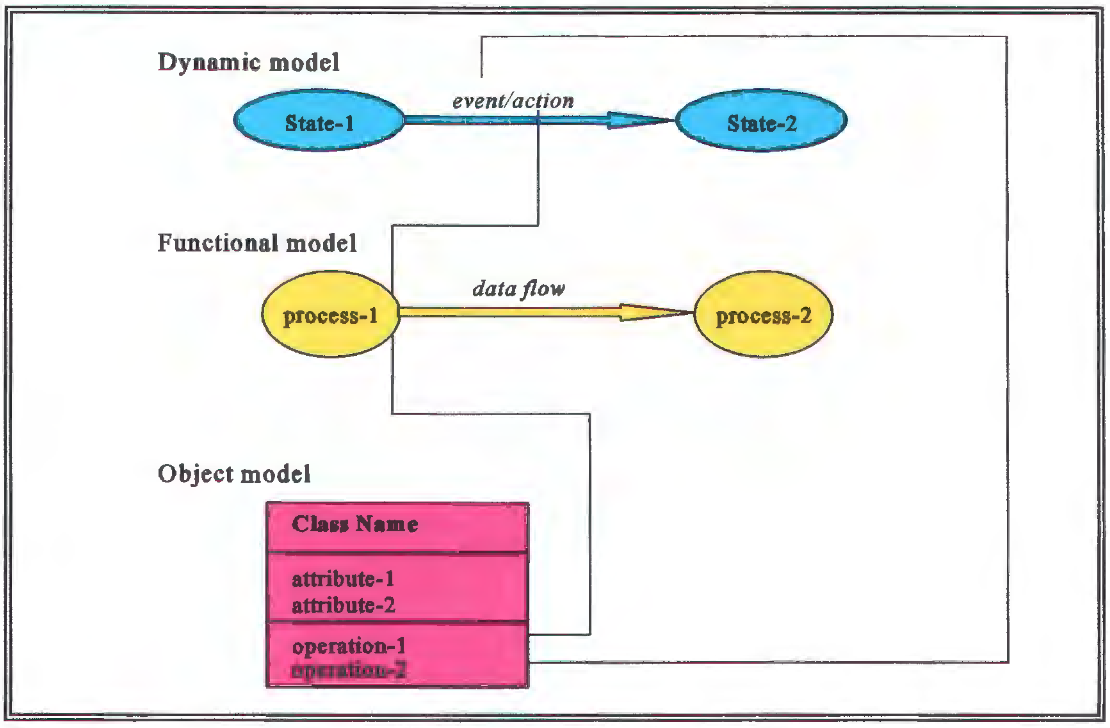


Figure 4.5 The relationships between the modeling primitives

Relative to the functional model:

- The object model explains the *structure* of the actors, data stores and data flows in the functional model. The operations in the object model correspond to the functions performed in the functional model.

- The dynamic model explains the *sequence* in which processes are performed.

Relative to the object model:

- The functional model explains the *operations* on the classes and the arguments of each operation. It therefore explains the supplier-client relationship among classes.
- The dynamic model explains the *states* of each object and the operations which are performed as it receives *events* and changes state.

Relative to the dynamic model:

- The functional model explains the *definitions* of the leaf actions and *activities* which are undefined with the dynamic model.
- The object model explains *what* changes state and undergoes operations.

4.5 The OMT Methodology for Design

The Design Cycle shown in *Figure 2.1* as a task on the Universal Level, is detailed in *Figure 4.6* by the author on the Worldly Level in DesignNet notation. The Worldly Level depicts the resources, the main tasks and deliverables, as well as the status of each design task for middle management, of a development project. As was explained in Chapter 2, the DesignNet notation is a structured, PetriNet based notation and was interpreted for object-oriented development according to the spiral model by Van der Walt (1994). The design tasks in *Figure 4.6* are System Design, Analyze and Evaluate Design Alternatives, Object Design and Implementation Planning. The proposed knowledge-based support which a design expert may offer are formulated for the design steps within each design task.

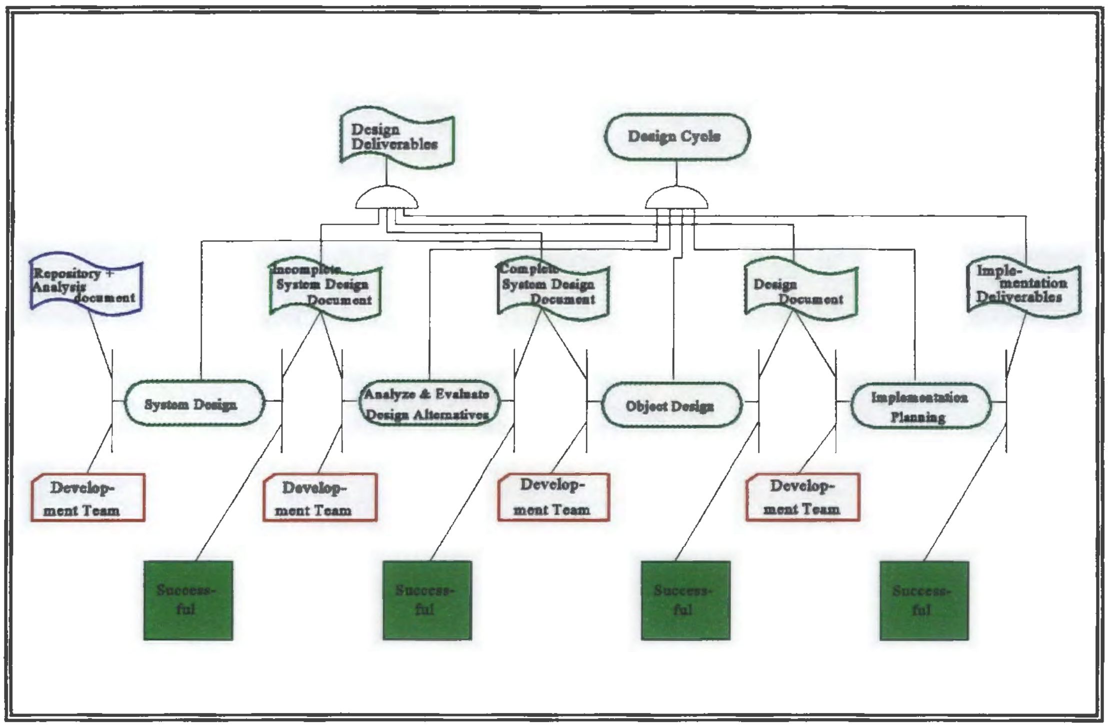


Figure 4.6 The Worldly Level of the Design Cycle

The design process starts with the deliverables of the Analysis Cycle as seen in the *Analysis Document*. These are:

i) a Problem Statement;

ii) an Object Model (= object model diagram + data dictionary) representing the *static structure* of the real-world problem. The object model diagram is supplemented by an abbreviated textual description including the purpose and scope of each entity;

iii) a Dynamic Model (= state diagrams + global event flow diagram) representing the behavior of each active object of the system in the form of a set of state diagrams;

iv) a Functional Model (= data flow diagrams + constraints) representing the functional derivation of values in the form of a levelled set of data flow diagrams.

System Design, as defined by the OMT Methodology, is redefined by the author and consists of a System Design task and an Analyze and Evaluate Design Alternatives task. During *System Design* a high-level strategy for solving the problem and building a solution is developed. Knowledge regarding the design method, the steps of the method, the representation schemata used and the deliverables of the design process is needed. The overall structure, style and organization of the system, which is the **system architecture**, is decided upon. The following steps are involved:

i) Organize the system into subsystems.
ii) Identify concurrency inherent in the problem.
iii) Allocate subsystems into processors and tasks.
iv) Choose an approach to management of data stores.
v) Handle access to global resources.
vi) Choose the implementation of control in software.
vii) Handle boundary conditions.
viii) Set trade-off priorities.

The first five steps belong to the System Design task and the last three steps belong to the Analyze and Evaluate Design Alternatives task. Each of these steps are considered next.

4.5.1 System Design

The five steps of the System Design task and the three steps of the Analyze and Evaluate Design Alternatives task are described next. Together these eight steps form System Design according to the OMT Methodology.

Step 1 - Organize the system into subsystems (Step 1 of the System Design task)
Divide the system into a small number of members (20 is probably too many). Each major member of a system is called a subsystem. Each subsystem encircles perspectives of the system which share some common grounds, for example the same physical location, similar functionality, or execution on the same kind of hardware. A subsystem is a bundle of classes, associations, operations, events, and constraints which have a well-defined and small interface (low coupling) with other subsystems. Each subsystem may in turn be divided into smaller subsystems of its own. The lowest level subsystems are called modules. A subsystem is usually identified by the services it provides, for example I/O processing, drawing pictures, or performing arithmetic.

The relationship between two subsystems may be of the form: *Client-supplier* or *peer-to-peer.* The client-supplier relationship refers to a relationship where the client calls on the supplier to perform some service and to reply with a result. In this relationship the client must know the interfaces of the supplier. The supplier, however, does not have to know the interfaces of its clients because the clients initiate the interactions using the supplier's interface.

In a peer-to-peer relationship each of the suppliers may call on the others. This is a more complicated interaction because the subsystems must know each other's interfaces. This kind of communication is also not necessarily followed by an immediate response. If one does have a choice, go for the supplier-client relationship because it is easier to build, understand and change a one-way interaction than a two-way interaction.

The breakdown of systems into subsystems may be ordered as an arrangement of horizontal *layers* or vertical *partitions*. Each layer defines its own theoretical environment, which may differ completely from other layers. Each subsystem recognize the layers below it, but has no information about the layers above it. A supplier-client relationship exists between lower layers, which are providers of services, and upper layers which are users of services. The layered architecture may be subdivided into two forms, namely *closed* and *opened*. When each layer is built only in terms of the immediate lower layer, it is called a closed architecture. Dependencies between layers are reduced and because a layer's interface only affects the next layer, changes are made easily. When a layer may use features of any lower layer to any depth, it is called an open architecture. The redefinition of operations at each level is reduced, which results in a more efficient and compact code. Open architecture does not comply with the principle of information-hiding. Changes to a subsystem may affect any higher subsystem. When choosing between the two kinds of architectures, the system designer must weigh up the relative value of efficiency and modularity. When a system is built in layers, it may be ported to other hardware/software systems by rewriting one layer. For this reason it is a good practice to introduce at least one layer of abstraction between the application and any services provided by the operating system or hardware. For example, define a layer of interface classes providing logical services (for example I/O services) and map them onto the concrete services which are system-dependent (for example I/O services for UNIX).

Partitions divide a system vertically into several low coupled subsystems. Each of these subsystems provides one kind of service. When subsystems have some knowledge of each other, but this knowledge is not deep (for example, *virtual memory management* and *a file system* in a *computer operating system*), one doesn't need to create major design dependencies, which means that vertical partitions may be used.

This step requires certain specific skills and types of knowledge:

- The ability to structure the system into subsystems by following a normative approach based on a specific knowledge of a structuring criterion. The decision to structure the system into subsystems is made by analyzing the object model according to the following criteria:
    - Identify object classes which execute on the same kind of hardware.
    - Identify object classes which execute in the same physical location.
    - Identify object classes with similar functionality.
- Knowledge of the semantics and syntax of the representation

schemata of the methodology used to develop the information system, such as the class and object diagrams of the object model as illustrated in *Exhibit 4.4*; or the event trace diagram and the state diagram of the dynamic model as illustrated in *Exhibit 4.5*; or the data flow diagram of the functional model as illustrated in *Exhibit 4.6*.

- Knowledge of the application domain (e.g. banking), and domain of discourse (e.g. an ATM system).

The knowledge and skills for this structuring step, as well as the other steps of design, are used to organize the knowledge base (explained in Section 4.6).

Experience has indicated that a combination of layered partitions and partitioned layers may be used in dividing a system into subsystems. *Figure 4.7* shows a block diagram of a typical application.

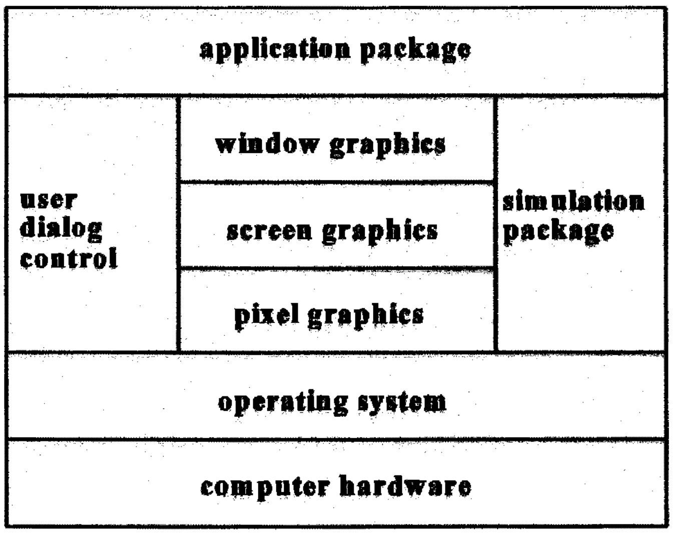


Figure 4.7 Block diagram of a typical application (Rumbaugh *et al.*, 1991)

Step 2 - Identify concurrency inherent in the problem (Step 2 of the System Design task)

At this stage the system designer must identify which objects must be active concurrently and which objects have activity that is mutually exclusive. Objects which have activity that is mutually exclusive, may be gathered together in a single *thread of control* or *task*. A *thread of control* is a path through a set of state diagrams on which only a single object at a time is active. When objects cannot be active together, they may be implemented on a single processor.

When two objects receive events at the same time, without interacting, they are *inherently concurrent.* If the events are unsynchronized, the objects cannot be gathered onto a single thread of control. Two subsystems which are inherently concurrent need not necessarily be implemented as separate hardware units. Logical concurrency in a uniprocessor may be simulated with hardware interrupts, operating systems and tasking mechanisms.

Identifying concurrency is done on the dynamic model. Examining state diagrams of individual objects as well as the exchange of events amongst them, may cause objects to be gathered together onto a single thread of control.

A thread is active within a state diagram (of an individual object) until an object sends an event to another object and waits for another event. The thread proceeds to the receiver of the event until it eventually returns to the original object. The thread splits if the object sends an event and continues executing. On each thread of control, only a single object at a time is active. Threads of control are implemented as *tasks* in computer systems.

The following types of knowledge are required:

- An interpretation of the constraints as specified in the requirements statement, and the resulting interdependence of objects.
- An interpretation of the arrival of events (as seen on an event trace diagram depicted in *Exhibit 4.5*) at objects and the resulting actions, i.e. whether or not such objects interact. If they do not interact, the objects are inherently concurrent.
- An analysis of the thread of control (on the path through a set of state diagrams on which only a single object at a time is active). *Exhibit 4.5* shows a meta model for a state diagram.

Step 3 - Allocate subsystems to processors and tasks (Step 3 of the System Design task)

The system designer must allocate each concurrent subsystem to a hardware unit. The hardware unit may be either a general purpose processor or a specialized functional unit. To be able to do this allocation the system designer must:

- *Estimate performance needs and the resources needed to satisfy them.* When needing more performance than that which a single CPU may provide, multiple processors or hardware functional units may be used. Estimating the required CPU processing power means computing the steady state load as the product of the number of transactions per second and the time required to process a transaction. The estimate should be increased to allow for an acceptable rate of failure due to insufficient resources.
- *Choose hardware or software implementation for subsystems.* Object-

orientedness makes its possible to see each hardware device as an object which operates concurrently with other objects which, in this case, may be other devices or software. The decision on which subsystems must be implemented in hardware and which in software must now be made by the system designer. For example, it is easier to buy a floating point chip than to implement floating point in software.

- *Allocating tasks to processors to satisfy performance needs and minimize inter-processor communication.* Tasks for software subsystems are assigned to processors because:
  - Certain tasks are required at specific physical locations, for example when a workstation needs its own operating system to enable operation when the inter-processor network is down.
  - Response time or information flow rate exceeds the available communication band-width between a task and a piece of hardware. For example, high-performance graphics devices have a high internal data generation rate. These devices require tightly-coupled controllers.
  - Computation rates are too high for a single processor. To minimize computation costs, subsystems which interact the most should be assigned to the same processor while independent subsystems are assigned to separate processors.

- *Determine the connectivity of the physical units which implement the subsystems.* At this stage the kinds and relative numbers of the physical units have been determined. The system designer must now choose the arrangement and form of the connections among the physical units. Make the following decisions:

- o Choose the topology of connecting the physical units.
- o Choose the topology of repeated units, for example when several copies of a particular kind of unit are included for performance reasons.
- o Choose the form of the connection channels and the communication protocols.

This step may be supported by the following "knowledge-based guidance":

- Estimate the required CPU processing power.
- Identify which subsystems will be implemented in hardware and which in software.
- Allocate the tasks for various software subsystems to processors.
- Determine the arrangement and form of the connections among the physical units.

Step 4 - Choose an approach for management of data stores (Step 4 of the System Design task)

The separation points of subsystems within an architecture may be provided by internal and external data stores. The general implementation of internal data stores consists of memory data structures and the general implementation of external data stores consists of files and/or databases. For example, an accounting system may use a database and files to connect subsystems. Files are a cheap, simple and permanent form of data store. Databases provide a higher level of abstraction than files but they are more complex and more expensive than files.

This step may be supported by the following "knowledge-based guidance":

- Identify the internal and external data stores.

Step 5 - Handle access to global resources (Step 5 of the System Design task)
Global resources must be identified and the system designer must also determine mechanisms for controlling access to them. The following are examples of global resources: Physical units, such as processors, tape drives and communication satellites; space, such as disk space, a workstation screen and the buttons on a mouse; logical names, such as object IDs, filenames and class names; and access to shared data, such as databases. A physical object may control itself by establishing a protocol for obtaining access within a concurrent system. A logical entity, for example filenames and databases, has the danger of conflicting access in a shared environment. This happens for example, when independent tasks simultaneously use the same filename. In this case each global resource must be owned by a "guardian object" which controls access to it. A "guardian object" may control more than one resource. The purpose of "guardian objects" is to place locks on subsets of a resource; serialize all access to a resource; and partition global resources into separate subsets which are managed at a lower level.

This step may be supported by the following "knowledge-based guidance":

- Identify global resources and determine mechanisms for controlling access to them.

Step 6 - Choose the implementation of control in software (Step 1 of the Analyze and Evaluate Design Alternatives task)

In a software system there are two kinds of control flow, namely external control and internal control. The flow of externally visible events among the objects in the system is called the *external control.* The three kinds of control for external events are procedure-driven sequential, event-driven sequential, and concurrent control. *Procedure-driven sequential* control refers to procedures issuing requests for external input, waiting for it, and when input arrives, control proceeding within the procedure which made the call. *Event-driven sequential* control occurs when control lives within a dispatcher or monitor provided by the language, subsystem, or operating system. Application procedures are now joined to events and are called by the dispatcher when the matching events occur. In a *concurrent* system, control lives concurrently in several independent objects, where each is a separate task.

The flow of control within a process is *internal control.* The three kinds of control flow which are used are procedure calls, quasi-concurrent inter-task calls, and concurrent inter-task calls. *Quasi-concurrent inter-task calls*, for example co-routines and lightweight processes, are programming facilities in which multiple address spaces or call stacks exist but in which only a single thread of control may be active at one time.

This step may be supported by the following "knowledge-based guidance":

- Choose a single control style for external events and control within a process.

Step 7 - Handle boundary conditions (Step 2 of the Analyze and Evaluate Design Alternatives task)

When talking about boundary conditions, the following must be addressed: *Initialization*, *termination* and *failure*. When the system is changed from a passive initial state to a supportive steady state condition, it is called *initialization*. *Termination* is the opposite of initialization, and is also simpler. Many internal objects may simply be abandoned and all the external resources which the task had reserved, must be released. *Failure* is the unplanned termination of a system. The ideal is to plan for a controlled exit. This means leaving the remaining environment as orderly and tidy as possible and logging or printing as much information about the failure as possible before terminating.

This step may be supported by the following "knowledge-based guidance":

- Verify that initialization will be handled.
- Verify that termination will be handled.
- Verify that failure will be handled.

Step 8 - Set trade-off priorities (Step 3 of the Analyze and Evaluate Design Alternatives task)

It is the work of the system designer to set priorities which will be used to guide trade-offs during the rest of the design. When desirable but incompatible goals are the issue, it is the system designer who has to set the priorities. For example, we need the system to be faster, for which we need extra memory, but we cannot afford a lot of extra memory. Where is the trade-off? Priorities are hardly ever definite. For example, trading memory for speed does not mean that any increase in speed, no matter how small, is worth any increase in memory, no matter how large. The system designer must remember that all the trade-offs are not made during system design, but establishing the priorities occurs at this stage. The trade-off decisions during subsequent Design Cycles will now be compatible.

This step may be supported by the following "knowledge-based guidance":

- Set priorities which will guide trade-offs during the rest of the Design Cycle.

At the end of the Analyze and Evaluate Design Alternatives task, the *System Design Document* is produced which describes the structure of the basic architecture for the system as well as high-level strategy decisions. *Figure 4.8* shows the System Design task and the Analyze and Evaluate Design Alternatives task.

After System Design, which involves the System Design task and the Analyze and Evaluate Design Alternatives task, the system designer must start with the *Object Design* task of the strategy chosen during System Design. During the Object Design task, the system designer elaborates on the analysis model and provides a detailed basis for implementation. This task involves the system designer in following these steps:

i) Combine the three models to obtain operations on classes.
ii) Design algorithms to implement operations.
iii) Optimize access paths to data.
iv) Implement control for external interactions.
v) Adjust class structure to increase inheritance.
vi) Design associations.

vii) Determine object representation.

viii) Package classes and associations into modules.

The eight steps of Object Design, as defined by the OMT Methodology, correspond to the eight steps of the Object Design task, as defined by the author. Each of these steps are considered next.

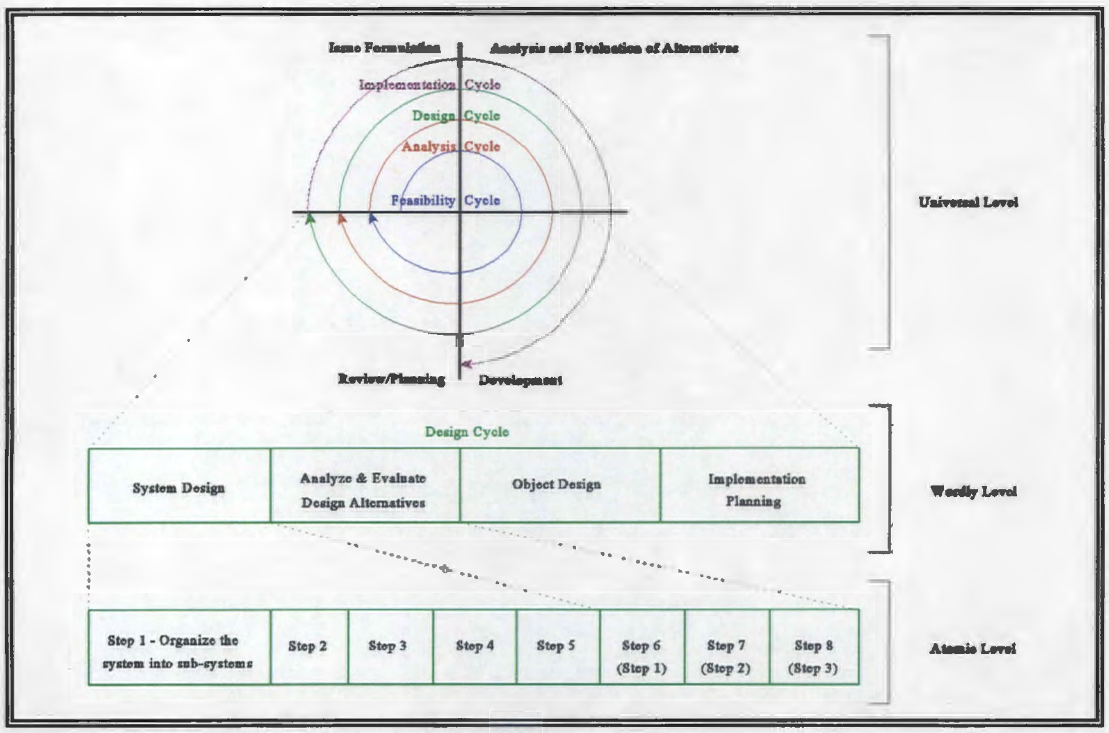


Figure 4.8 Modeling System Design in the Design Cycle

4.5.2 Object Design

The eight steps of Object Design follow.

Step 1 - Combine the three models to obtain operations on classes

The input to the object design task comprises the object, dynamic and functional models. At this stage the actions and activities of the dynamic model and the processes of the functional model must be converted into operations, attached to classes, in the object model. The first step will be to define an operation for each event in the dynamic model. The second step will then be to find an operation for each data flow diagram in the functional model. The processes in the data flow diagram constitute sub-operations.

This step may be supported by the following "knowledge-based guidance":

- Identify an operation for each event in the dynamic model.
- Identify an operation for each data flow diagram in the functional model.

Step 2 - Design algorithms to implement operations

Each operation must now be formulated as an algorithm. The algorithm designer must:

- Choose algorithms which minimize the cost of implementing operations. Concentrate on:
  - o *Computational complexity*, for example how does processor time increase as a function of the size of the data structures?
  - o *Ease of implementation and understandability*, for example give up some performance on non-critical operations if they may be implemented quickly with a simple algorithm.
  - o *Flexibility*, for example when an algorithm is highly

optimized, it is often difficult to read and change and this may force one to provide two implementations of critical operations so that the simple and inefficient algorithm may be implemented quickly and used to validate the system. Then the complicated and efficient algorithm's correctness may be validated against the simple one's correctness.

  - o *Fine-tuning the object model*, for example if the object model were structured differently, would there be other alternatives?

- Select data structures appropriate to the algorithms. Such data structures include arrays, lists, queues, stacks, sets, bags, dictionaries, associations, trees and many variations on these, such as priority queues and binary trees.
- Define new internal classes and operations as necessary. During the development of algorithms, new classes of objects may be needed to hold intermediate results. New, low-level operations may be invented during the decomposition of high-level operations.
- Assign responsibility for operations which are not clearly associated with a single class. Most operations have obvious target objects. Some operations may be performed at several places in an algorithm, by one of several objects, as long as they eventually get done. When more than one object is involved in an operation, one must decide which object plays the lead role in the operation in order to be able to decide which class owns this operation.

This step may be supported by the following "knowledge-based guidance":

- Formulate an algorithm for each operation specified in the functional model.

Step 3 - Optimize access paths to data

The analysis model represents the logical information of a system. It is semantically correct but insufficient. The design model must now add detail to support efficient information access. An optimized system is obscured and camouflaged to a greater degree and is less likely to be re-usable, and it is the task of the system designer to find an appropriate balance between efficiency and transparency. During design optimization the system designer must:

- Add redundant associations to minimize access cost and maximize convenience. For example, he must provide indexes for recurrent and expensive operations with a low hit ratio because such operations are wasteful to implement using nested loops.
- Re-arrange the computation for greater efficiency. Narrow the search as soon as possible by eliminating dead paths as early as possible. For example, suppose we want to find all employees who speak both Afrikaans and English. Suppose 5 employees speak English and 200 speak Afrikaans. It will be better to test and find the English speakers first, then test if they speak Afrikaans.
- Save derived values to avoid re-computation of complicated expressions. This information may be retained in new objects or classes which must be defined. The class which holds the cached data must be updated if any of the objects on which it depends are changed.

This step may be supported by the following "knowledge-based guidance":

- Identify redundant associations which will minimize access cost and maximize convenience.
- Verify the necessity for the re-arrangement of computation for greater efficiency.
- Identify derived attributes to be saved to avoid re-computation of complicated expressions.

Step 4 - Implement control for external interactions

During system design, a strategy was decided on for realizing the dynamic model. This strategy must now be followed. To implement the dynamic model there are three different basic approaches:

- Using the location within the program to hold state (procedure-driven system).
- Direct implementation of a state machine mechanism (event-driven system).
- Using concurrent tasks.

This step may be supported by the following "knowledge-based guidance":

- Identify the representation of control within a program which may be:
  - Where the location of control within a program implicitly defines the program state,
  - explicitly representing and executing state machines,
  - where an object may be implemented as a task in the programming language or operating system.

Step 5 - Adjust class structure to increase inheritance

The system designer should:

- Rearrange and adjust classes and operations to increase inheritance. Sometimes operations in different classes are alike but not identical. If the definitions of the operations or the classes are slightly adjusted, the operations may often be made to match so that they may be covered by a single inherited operation.
- Abstract common behavior out of groups of classes. By doing this a common superclass may be created which implements the abstracted shared features, leaving only the specialized features in the subclasses.
- Use delegation to share behavior where inheritance is semantically invalid. When an existing class already implements some of the behavior which we want to provide in a newly defined class, but in all other respects the two classes are different, the system designer must not inherit from the existing class. Rather make the one class an attribute or associate of the other class. Now one object may selectively invoke the desired functions of another class, using delegation rather than inheritance.

This step may be supported by the following "knowledge-based guidance":

- Identify the rearrangement and adjusting of classes and operations to increase inheritance.
- Identify common behavior in groups of classes.
- Verify when inheritance will be semantically invalid and use delegation to share behavior.

Step 6 - Design associations

Associations provide access paths between objects. A strategy must be formulated for implementing the associations in the object model. The following steps must be taken:

- Analyze the traversal of associations. If an association is only traversed in one direction, it may be implemented as a pointer. Associations may also be traversed in both directions and implemented using three different approaches:
  - Implement as an attribute in one direction only and when a backward traversal is required, then perform a search (when minimizing both the storage cost and the update cost is important and also if there is a big difference in traversal frequency in the two directions).
  - Implement as attributes in both directions (when accesses outnumber updates).
  - Implement as a distinct association object, independent of either class (when expanding predefined classes from a library which cannot be altered, because the association object may be added without adding any attributes to the original classes).
- Implement each many-to-many association as a distinct class, in which each instance represents one link and its attributes.

This step may be supported by the following "knowledge-based guidance":

- Identify a strategy for implementing the associations.

Step 7 - Determine the exact representation of object attributes

The system designer may use primitive types in representing objects or he may combine groups of related objects. The system designer must make this choice. Classes may be defined in terms of other classes, but eventually everything must be implemented in terms of built-in primitive data types, such as integers, strings and enumerated types.

This step may be supported by the following "knowledge-based guidance":

- Identify when to use primitive types in representing objects and when to combine groups of related objects when implementing objects.

Step 8 - Package classes and associations into modules

Packaging is important to permit different persons to work together on a program without affecting one another's work. Packaging involves:

- Hiding internal information from outside view. This permits implementation of a class to be changed without requiring any clients of the class to adjust code.
- Coherence of entities. When entities (e.g. classes, operations and modules) are organized according to an agreeable plan and all its parts fit together to achieve a common goal, such an entity is coherent.
- Constructing physical modules. The interfaces of modules must be small and well-defined. In the same module one must find classes which are closely connected by associations. Modules should have some practical cohesiveness or harmony of purpose. Classes in a

module should represent similar things in the application. Encapsulate strong coupling within a single module.

This step may be supported by the following "knowledge-based guidance":

- Identify and establish physical packaging.

A *Design Document* will now be constructed which consists of:

i) a Detailed Object Model plus

ii) a Detailed Dynamic Model plus

iii) a Detailed Functional Model.

4.6 The Organization of the Design Knowledge Base

The objective of this knowledge-based system for design is to ease the task of the designer to achieve a good design and to avoid possible pitfalls and errors. The Design knowledge base is organized into a number of separate aspects, each containing rule sets providing support for a specific perspective of design. Rule-based expert systems capture the knowledge of a domain expert in sets of rules. These enable one to reason about a specific problem at hand. The Kappa-PC System, the chosen knowledge-based environment for this investigation, utilizes rule sets to formalize the knowledge-based guidance, in this case for the Design Cycle of the SDLC when following the OMT approach. The "knowledge-based guidance" which was formulated in Section 4.5 is reformulated as questions (that may lead to rules) in rule sets. These questions assist the system designer in making design decisions regarding the target system, via an object-oriented design process as shown in *Figure 4.9.* (The target system is the system under development for the purpose of implementing it.) The different sets of rules differentiate between rules regarding different areas. It was decided that the rules will be structured in the following rule sets: The Work-break-down Structure Rule Set and the Deliverables Rule Set for the Methodological Aspect; the System Design Rule Set and the Object Design Rule Set for the Design Aspect; the Design Verification Rule Set and the Re-usability Rule Set for the Quality Assurance and Verification Aspect; a rule set for the Consistency and Completeness Aspect; and a rule set for the Prototyping Aspect. Each of these rule sets are reviewed next. Emphasis is placed on the Methodological Aspect and the Design Aspect. The Quality Assurance and Verification Aspect, the Consistency and Completeness Aspect and the Prototyping Aspect are not discussed in detail, but are mentioned for the sake of completeness.

Structure of Rules

The general structure of the rule sets into aspects is presented here, with selected examples to illustrate the questions that could lead to rules. These questions are phrased in English syntax for purposes of user-friendliness. These questions in general provide guidance with typical problem areas for an inexperienced OMT designer.

- ***Methodological Aspect***

  The aspect concerns knowledge regarding the design method, the steps of the method, the representation schemata used and the deliverables of the design process.

  - ***Work-break-down Structure Rule Set***

    This rule set contains all the rules which have to do with the step-by-step following of the methodology. For example: "*First do System Design before commencing to Object Design.*"

- ***Deliverables Rule Set***

  A rule set which represents the rules concentrating on deliverables. For example:

  "*Before starting with Design, do all the analysis deliverables exist?*"

  "*Can secondary deliverables be automatically produced from primary deliverables?*"

  "*Can the completeness of the deliverables be checked and verified?*"

  "*Is cross-reference of deliverables supplied?*"

- ***Design Aspect***

  The aspect concerns the syntax and semantics of the three conceptual models, namely the object model, the functional model and the dynamic model. The meta models of the object model, dynamic model and the functional model are presented in *Exhibits 4.1*, *4.2* and *4.3* respectively. The meta models for class/object diagrams, an event trace diagram and a state diagram, and a data flow diagram are presented in *Exhibits 4.4*, *4.5* and *4.6* respectively.

  - ***System Design Rule Set***

    Structuring mechanisms applied to the object model in Step 1 of the System Design task. Each of the object classes in the object diagram is queried by means of a dialogue to determine adherence to specific criteria. For example:

"*On what kind of hardware component does object-class-A execute?*"

"*In which physical location does object-class-A execute?*"

"*Identify the functionality of object-class-A.*"

In Step 2 of the System Design task, the dynamic model, the event trace diagram and the state diagrams form the basis for an analysis of concurrent behavior. Since the investigation concentrated on applications with limited dynamics, this part of the knowledge base is not explained further here.

- o ***Object Design Rule Set***

  This comprises all the rules which have to do with the step-by-step following of Object Design, as discussed in Section 4.5.2. For example:

  "*Identify an operation for each data flow diagram in the functional model.*"

- ***Quality Assurance and Verification Aspect***

  This aspect concentrates on the quality assurance and verification of the Design Cycle.

  - o ***Design Verification Rule Set***

    The rules which concentrate on "Is this a good design?", falls under this rule set. For example:

    "*Are there other behaviors for an object which should be included but which were not explicitly stated in The*

Requirements Specification?"

- ***Re-usability Rule Set***

  This rule set represents all the rules which point re-usable components out or the rules which gather re-usable components. For example:

  "Does a re-usable component exist that will be able to structure a document describing the current available deliverables of the Design Cycle?"

- ***Performance Rule Set***

  This rule set represents all the rules which have to do with the performance, for example:

  "Identify performance constraints and verify their relevance."

- ***Consistency and Completeness Aspect***

  Independent rules which check for consistency and completeness will be gathered under this aspect. For example:

  - *"Are all the steps in the System Design task completed before one starts with the Object Design task?"*
  - *"Verify consistency of high-level strategy constraints."*
  - *"Verify that transition from analysis to design is consistent (i.e. that only the appropriate design details are added)."*
  - *"Determine whether design detail is sufficient to proceed to implementation."*

- o *"Verify that design detail exists for scheduled tasks of Project Management Planning."*
- o *"Is there an inverse (undo) operation, or a complementary operation which is an appropriate addition to an object's protocol?"* (for example: For every ADD-operation, is there a DELETE-operation and a MODIFY-operation?).

- ***Prototyping Aspect (during Design)***

  The rules which represent prototyping will be gathered here. For example:
  - o *"At what point during the Design Cycle is prototyping possible?"*
  - o *"At what point during the Design Cycle is prototyping desirable?"*

4.7 Conceptual Model of Proposed Solution

The rules for the Design Cycle which were discussed in Sections 4.5 and 4.6 form part of a knowledge base. With reference to *Figure 1.1*, and the method of investigation outlined in Section 1.4.1, the block diagram in *Figure 4.9* is an instance of the conceptualization of the proposed solution. The object-oriented design steps are supported by this knowledge base and the knowledge-based environment which are discussed in the next chapter.

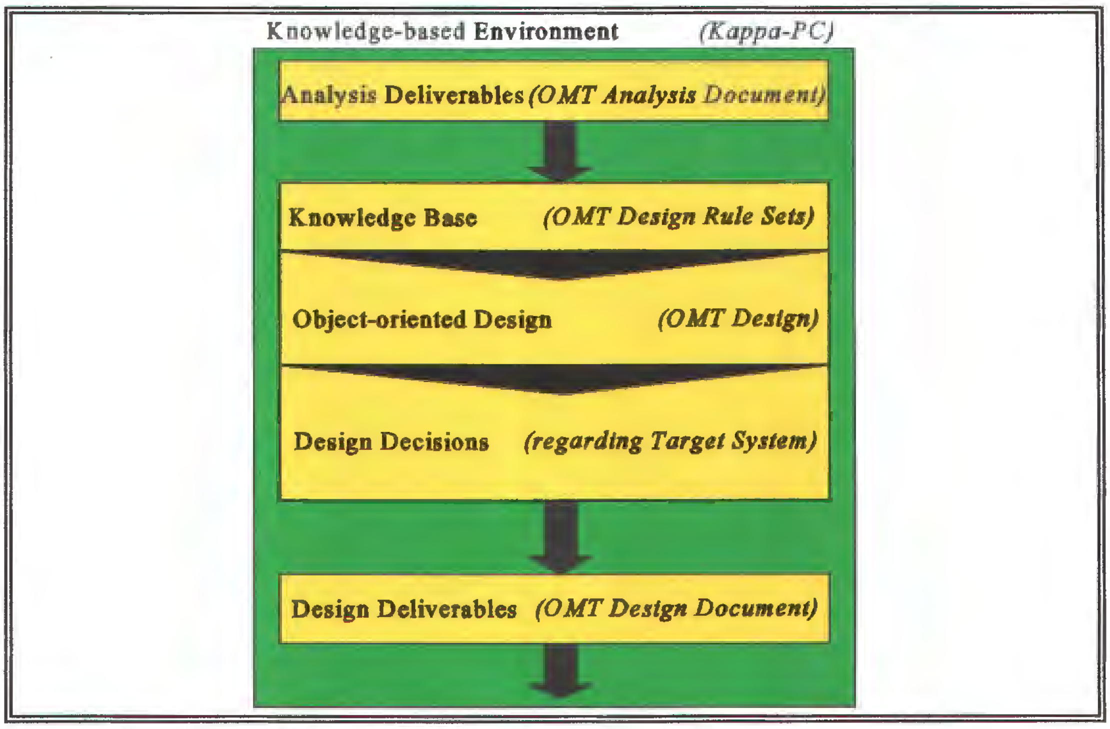

Knowledge-based Environment (Kappa-PC)
Analysis Deliverables (OMT Analysis Document)
Knowledge Base (OMT Design Rule Sets)
Object-oriented Design (OMT Design)
Design Decisions (regarding Target System)
Design Deliverables (OMT Design Document)

Figure 4.9 Object-oriented Design within a Knowledge-based Environment

4.8 Summary and Conclusion

Modeling in general was discussed with emphasis on the clarity which it offers regarding the requirements of the application.

Rumbaugh's OMT approach is described. The three models which form the basis of this methodology are explained separately. The three models are the object model, which shows the static data structure of the real world, the dynamic model, which shows the time-dependent behavior of the system and the objects in it, and the functional model, which shows how values are computed, without regard for sequencing, decisions, or object structure. A summary, which explains the relationships amongst the three models, follows.

The OMT approach for design is described in detail with emphasis on the two tasks of the Design Cycle. The first task is System Design where the overall architecture of the system is decided upon. The second task is Object Design where the proposed system moves to a detailed basis for implementation.

A structure of knowledge-based guidance is described and 5 different aspects are explained which will form one or more rule set/s each. The conceptual model of the proposed solution is depicted in a block diagram, illustrating the different building blocks of this solution.

CHAPTER 5
KAPPA-PC Knowledge-based Environment

CONTENTS

5.1 Introduction

In this chapter the knowledge-based environment of *Figure 4.9*, namely Kappa-PC will be described. This object-oriented environment was used to develop a prototype of a target system which serve to demonstrate concept, as was formulated in Chapter 4. One of the building blocks of the environment is an expert system with which the knowledge-based support is demonstrated. First the objects and methods for a knowledge base must be constructed. Secondly the system which specifies how objects should behave, or which may reason about the objects by using rules, is constructed.

5.2 Kappa-PC [1]Key Concepts

Kappa-PC consolidates five key concepts (Kappa-PC Quick Start Manual, 1992):

i) Object-oriented development

The primitives of the target system are represented by structures called *objects*. These objects may be either *classes* or *instances* of classes. A *hierarchy* is a structure which represents the relationships among objects. The processes of the target system are represented by *monitors* and *methods*. Application components which are developed by means of an object-oriented methodology are re-usable for new applications because they are independent entities.

ii) High-level descriptive language

Kappa-PC has its own descriptive language called Kappa-PC Application Language (KAL). The language has a set of 280 predefined functions and provides for fast prototyping, procedural programming and ample representation.

iii) High-performance rule systems

Rules and *goals* represent the criteria which one uses to make decisions. These decision criteria may easily be changed. By using rules-based reasoning one incorporates expertise, heuristics and rules of thumb into software solutions. Each rule specifies a set of conditions and a set of conclusions to be made if the conditions are true.

[1] The concepts used in this chapter are interpreted according to the authors of Kappa-PC, and may not correspond precisely to established interpretations found in the literature.

iv) Graphical development and delivery

A graphical development interface, including browsers, editors, layout tools, language interpreter and a debugger, makes building an application easier. The representation of the solution to the end-user may be done through a complete graphical interface (GI) of forms, images and dialogue boxes.

v) Database mapping

Existing data gets mapped into the application. The results of running the application may in turn be used to update the existing data.

These concepts form the basis for application development within the Kappa-PC environment. The architecture of the environment is depicted in *Figure 5.1*. It is an extension of a figure found in the Kappa-PC Quick Start Manual. Interaction with the knowledge base takes place by means of an application language, KAL, the end-user interface tools of the End-User Interface and the tools of the Developer Interface.

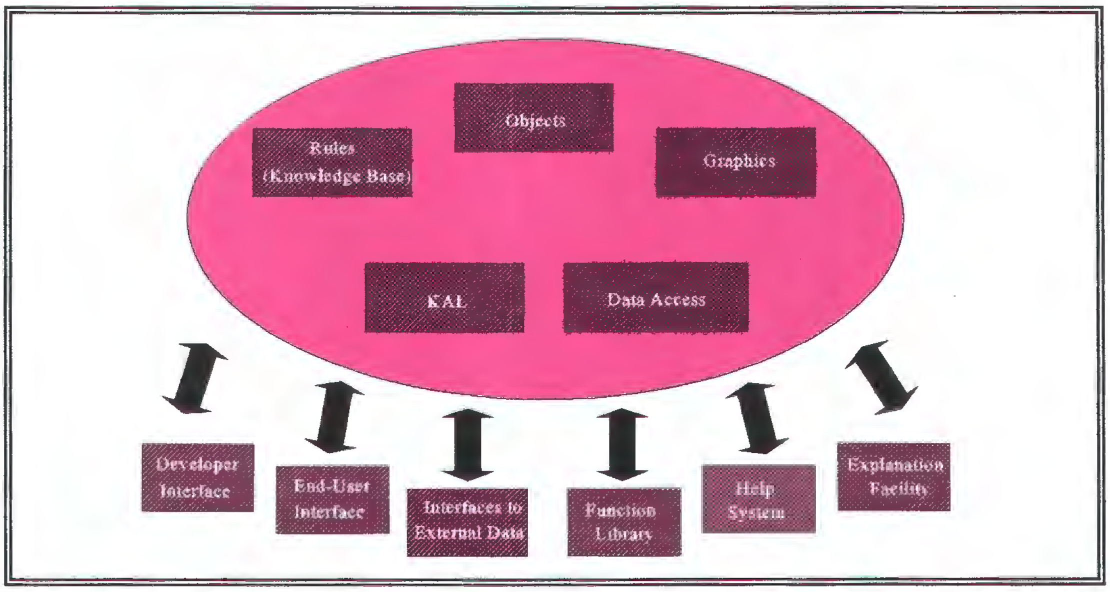


Figure 5.1 Kappa-PC Building Blocks

5.3 Kappa-PC Building Blocks

The building blocks are the modeling primitives within the environment. *Objects* are primary building blocks. Any target system may be viewed as a collection of objects (automobiles) with certain attributes (color, price), parts (doors, tires), abilities (moving, turning) and/or relationships to one another. *Classes* are categories of the knowledge base which share important characteristics. A class may be a group or collection of objects. For example, *Autos* is a class referring to all automobiles. *Subclasses* are subsets of another class. For example, *Sedans* and *StationWagons* are two subclasses of the class *Autos*. *Instances* are specific elements within knowledge base categories. It is a specific object, for example *Johan'sCar*. *Slots* are attributes of both classes and instances. Each slot describes a characteristic of the object. For example, an object representing a car could have a slot for color. *Red* may be the value of the *Color* slot. *Inheritance* exists between two classes or between a class and one of its instances. Inheritance illustrates the relationship between a class and its subclass. This hierarchy is called the *object hierarchy* (*Figure 5.2*). *Methods* define the "behavior" of specific objects. Methods are written in KAL and may be activated either by monitoring slots or by receiving messages. The technique of storing an object's behavior as one of its attributes is part of *object-oriented programming*. *Functions* perform the key tasks in the application development process. Kappa-PC provides a library of functions with which one may orchestrate the knowledge base. Using KAL (or the "C" language) one may build one's own functions as well.

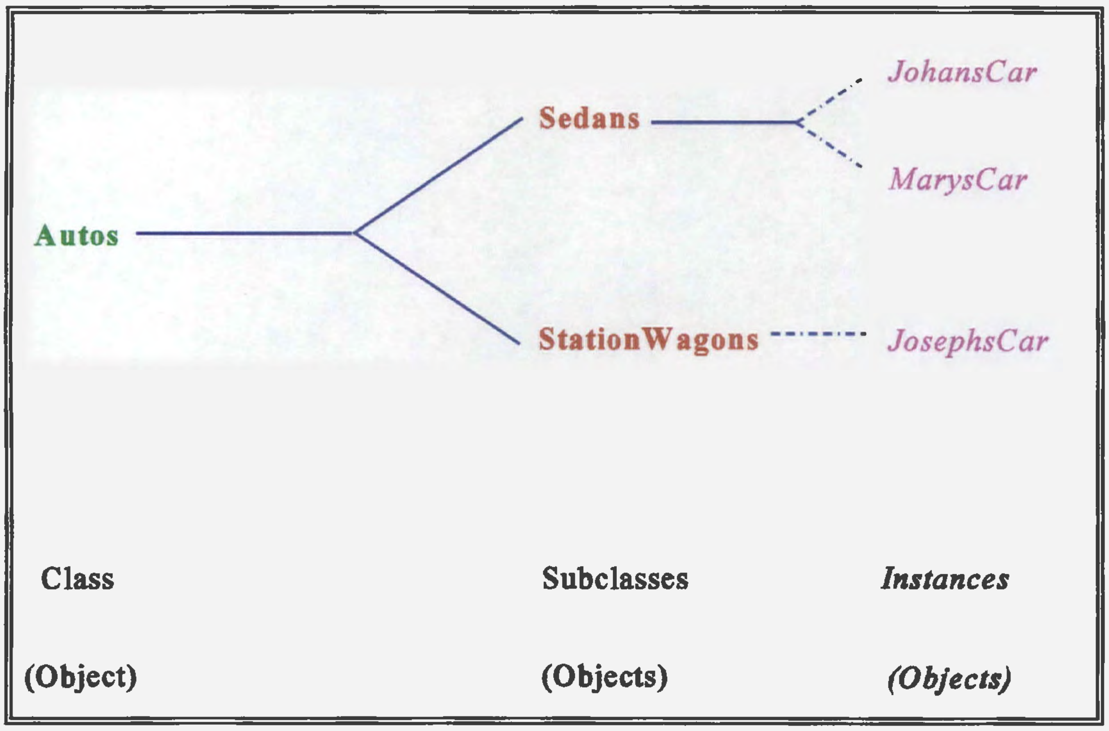


Figure 5.2 An Object Hierarchy (Kappa-PC User's Guide)

Monitors are private functions or functions which change the value of slots. *Images* are graphical representations of data or tools for changing data. With images one may create a user interface. *Rules* are If-Then statements which allows one to "reason" across the knowledge base. A rule specifies the conditions under which a particular action or inference may occur.

5.4 The KAL Language

KAL is Kappa-PC's application language (Kappa-PC User's Guide Manual, 1992) which one uses to write rules, methods and functions. It is also a language which one uses to add, delete and retrieve information from the knowledge base. KAL allows developers to create, control, modify, test, or delete the different application components, such as classes, instances, rules, goals, functions, end-user interface components and developer's interface components. KAL enables the developer to perform different operations. Some examples are mathematical computations *(Sin - calculates the sine value of an angle)*, list manipulation *(LengthList - gets the number of items in a list)*, string manipulation *(#= - compares two text strings)*, logical operations *(OR - checks if any of the argument values is TRUE)*, file input/output *(CloseReadFile - closes a file previously opened with the function OpenReadFile)*, and knowledge-processing functions *(ActivateRule - Adds a rule to the list of rules to be considered by the inference engine)*. KAL also assists in setting up a database mapping environment and read or write to various PC databases *([2]dBase)*, spreadsheets *([3]Lotus 1-2-3)*, SQL relational databases *([4]Sybase)*, or ASCII files. With KAL the developer may integrate his own user-defined functions within Kappa-PC.

5.5 End-user Interface

This interface provides the tools necessary to create a user-friendly application using windows, menus and other graphical techniques. These tools fall into three categories (Kappa-PC User's Guide Manual, 1992):

[2] dBase is a registered trademark of Ashton-Tate.

[3] Lotus and 1-2-3 are registered trademarks of Lotus Development Corporation.

[4] Sybase is a registered trademark of Sybase, Inc.

i) ***The control of the Kappa-PC windows***

There are eight standard windows in the Kappa-PC environment. They are Kappa-PC Main Window, Object Browser, Session Window, Edit Tools Window, KAL Interpreter, KAL View Debugger, Find/Replace, Rule Relation Window, Rule Trace Window and Inference Browser.

- Kappa-PC Main Window - This is an interface for managing the development of an application by saving and retrieving files and applications and by managing all the Kappa-PC windows.
- Object Browser Window - Allows one to view and modify all the objects and their relationships in an application. It presents one with a graphical view of the object hierarchy.
- Session Window - This is the main interface for the end-user of a Kappa-PC application.
- Edit Tools Window - This window provide access to all knowledge items in Kappa-PC - for example, classes, instances, functions, rules and goals.
- KAL Interpreter Window - Allows one to type in and interpret KAL expressions.
- KALView Debugger Window - Warns one about errors in function and method code.
- Find and Replace Window - One may find and replace text which appears anywhere in the knowledge base.
- Rule Relation Window - A graphical way of representing relationships between rules.
- Rule Trace Window - Allows one to view the rules which the inference engine invokes in the form of a transcript. One may also

follow the impact of reasoning on particular slots in the knowledge base.

- Inference Browser Window - One may view the rules which the inference engine invokes in the form of a graphical network. With this window one may see how the system arrived at its conclusions by examining its lines of reasoning once the reasoning process is complete.

ii) Pop-up windows provide user interaction

The application may interact with the user via pop-up dialogue windows. These windows pop up in the middle of the screen and demand the prompt attention of the user. The functions available for these windows are:

- PostMessage - This function allows one to present the user of the application with a simple message.
- SetPostMessageTitle - Changing the default title "KAPPA" to any user-defined title is possible with this function.
- AskValue - With this function one may present the user of the application with a standard user request form. The value of a single-valued slot must be entered.
- PostMenu - When one wants to present the user of the application with a list of options, use this function.
- PostInputForm - This function allows one to present the user of the application with a customized form for data input.

iii) The "Session Window" provide application graphics

The Session Window provides a medium for communication between an application and its user. This Session Window may be customized by the developer to change its appearance. The run-time (or delivery) version of Kappa-PC will essentially be this Session Window which has been customized to fit requirements. Individual graphics objects are known as *images*. Examples of images are line plots, bit maps, state boxes and meters. Some of the images display information to the end-user about the condition of the application. Some images allow the user to input information into the application. The types of images are as follows:

- Line plot image - Plotting up to six pairs of x-y vectors containing numerical values.
- Bit map image - Displaying a bit map file on the screen.
- State box image - Monitoring the value of a text slot while an application is running.
- Meter image - Monitoring the value of a numeric slot during the running of an application.
- Button image - A rectangular area which may activate a function when the mouse is clicked over it.
- Drawing image - Drawing a customized image.
- Edit image - Allows one to type in a value for a single-valued slot.
- Slider image - Entering a value into a single-valued slot which requires a numeric value.
- Text image - Displaying a fixed piece of text, such as a label or title.
- Transcript image - A text window into which one may output text at any time while running the application.

- SingleListBox - A listbox which may be attached to a slot with a single value. This is an input/output image where one may view and modify the data in the attached slot.
- MultipleListBox - A listbox which may be attached to a slot with multiple values. This is an input/output image where one may view and modify the data in the attached slot.
- RadioButtonGroup - A group of buttons which may be attached to a slot with a single value. This is an input/output image.
- CheckBoxGroup - Displays one checkbox for each permissible value defined in the OwnerSlot. One may change the value of the slot in the Session Window.
- CheckBox - Allows one to display the Boolean value of a single-valued slot as well as changing the value of the slot in the Session Window.
- ComboBox - Combining the editing ability of an Edit box with the display ability of a SingleListBox.

5.6 Developer's Interface

Kappa-PC has eight application development tools (Kappa-PC User's Guide Manual, 1992):

i) The *Object Browser* window which rapidly defines, creates, modifies, deletes, renames, hides and shows the object representation.

ii) The *Knowledge Tools* window providing access to knowledge editors. For example, class and instance editors, slot editors, slot option editors, method and function editors, as well as rule and goal editors.

iii) The *KAL Interpreter* window which prototypes applications.

iv) The *Session* window creates, manipulates and displays user-friendly dynamic displays for solution presentation.

v) The *Rule Relations* window dynamically displays rule networks and interdependent application relationships.

vi) The *Rule Trace* window traces chaining processes where the developer may step through inferencing one step at a time. The rule trace window is designed to assist during debugging.

vii) The *Inference Browser* window speeds up debugging and assists during this process by graphically displaying the inferencing process. It allows interactive editing of rules.

viii) Kappa-PC applications may also be developed with any ASCII text editor by using the "SAVE" and "RETRIEVE" facilities.

5.7 Interfaces to External Data Sources and Programming Languages

There are links from Kappa-PC to popular software (Kappa-PC Quick Start Manual, 1992). This helps to safeguard current software investments while adding additional functionality to existing applications. Kappa-PC may interface to databases (which may be dBase, [5]Ingres, Sybase, [6]INFORMIX, and [7]DB2),

[5] Ingres is a registered trademark of The ASK Group, Inc.

[6] INFORMIX is a registered trademark of INFORMIX Software Inc.

[7] DB2 is a registered trademark of International Business Machines Corporation (IBM).

spreadsheets (Lotus 1-2-3), graphics and Computer Aided Design (CAD) packages, conventional programming languages (such as FORTRAN, C, and PASCAL), and ASCII files.

5.8 Rule-based Reasoning

Rule-based reasoning allows developers to merge rules of thumb, heuristics, and knowledge typically acquired by experience or judgement (Kappa-PC User's Guide Manual, 1992). *Rules*, represented as "if" (conditions) and "then" (actions) statements, specify logical relationships among values of slots. Rules and object slots are compiled into a modified [8]Rete inference network. The *inference engines* are forward chaining, backward chaining, as well as the forward engine may be invoked during backward inferencing and vice versa. Forward chaining finds the consequences of known facts and the consequences of those consequences. Backward chaining tries to verify a fact by finding rules which may prove it. It also works on multiple conclusions. The *control mechanisms* are the goal, the agenda, and four rule-activating schemata: depth-first, breadth-first, best-first, and selective. The goal is an expression in the current knowledge base, representing a "quit" test or postulate to be verified. The agenda is a queue of object:slot pairs to be processed by the forward chainer. The last control mechanism is priorities, given to rules, for conflict resolution. *Figure 5.3* shows an instance of the conceptual model.

[8] Rete refers to a fast algorithm for the Many Pattern/Many Object Pattern Match Problem (Forgy, 1979).

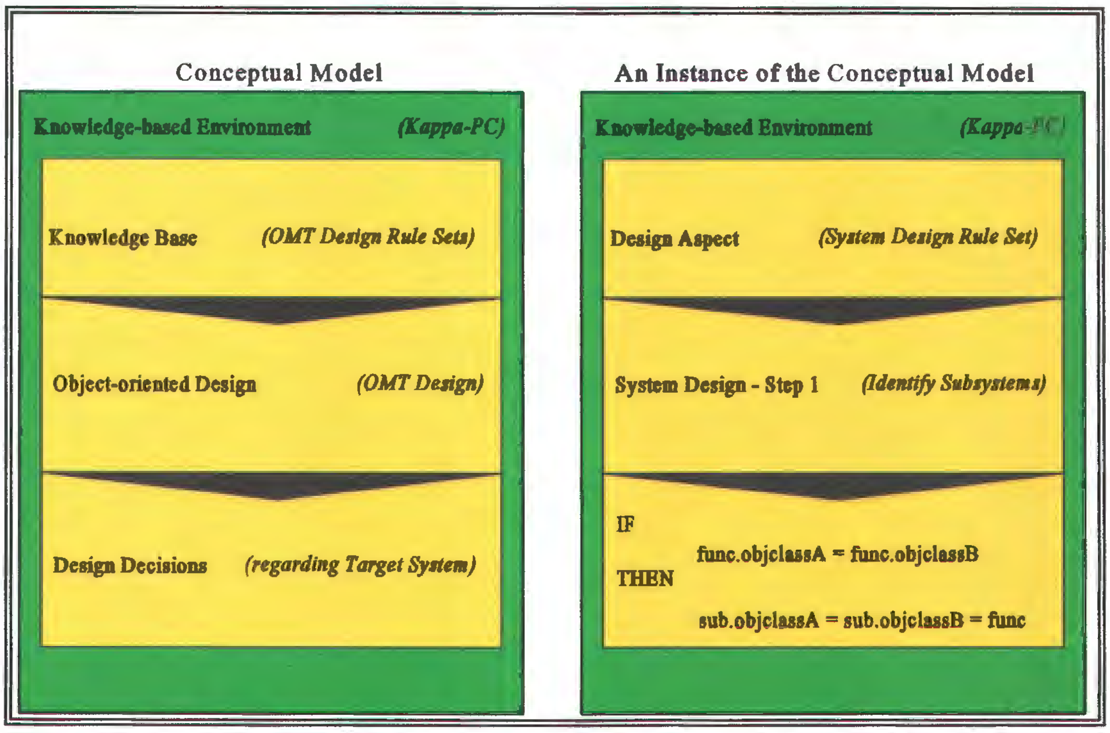

Conceptual Model
Knowledge-based Environment (Kappa-PC)
Knowledge Base (OMT Design Rule Sets)
Object-oriented Design (OMT Design)
Design Decisions (regarding Target System)
An Instance of the Conceptual Model
Knowledge-based Environment (Kappa-PC)
Design Aspect (System Design Rule Set)
System Design - Step 1 (Identify Subsystems)
IF
func.objclassA = func.objclassB
THEN
sub.objclassA = sub.objclassB = func

Figure 5.3 An Instance of the Conceptual Model

5.9 Summary and Conclusions

The Kappa-PC key concepts are explained, namely object-oriented development, a high-level descriptive language, a high-performance rule system, graphical development and delivery, and database mapping.

The Kappa-PC building blocks are identified and each are briefly described. They are objects, classes, subclasses, instances, slots, methods, functions, monitors, images and rules. The KAL Language is discussed and the importance of having a high-level development language is explained. The End-user

Interface is described in terms of all the tools. The tools of the Developer's Interface are discussed and the Interfaces to External Data Sources and Programming Languages are explained. The powerful rule-based reasoning of Kappa-PC is addressed and its building blocks, namely rules, inference engines, and control mechanisms are explained.

Kappa-PC is a powerful and user-friendly, object-oriented, application development environment. It has a strong expert system component and lends itself to a broad range of possible applications.

CHAPTER 6
The Design Prototype

CONTENTS

6.1 Introduction

In Chapter 5 the Kappa-PC knowledge-based environment was described. A design prototype was built to serve as a demonstration of the conceptual model formulated in Chapter 4 and depicted in Figure 5.3 of Chapter 5. The prototype focuses on aspects of the System Design task which is part of the Design Cycle and demonstrates that it is possible to support the Design Cycle of the software development life cycle (SDLC) with a knowledge-based system. The prototype does not claim to be a fully workable system but is a demonstration of limited scope and restricted functionality of the Design Cycle. Although the prototype could be enhanced to include explanations of guidance provided by the system and the relevant rules used, this was consciously omitted. A User's Manual, accompanying the prototype software, was compiled and is included as *Appendix F*. *Appendix G* explains the design step (Step 1) which is applied to an ATM problem and which is demonstrated within the customized Kappa-PC environment, and the source code of the prototype is in *Appendix H.* This chapter explains the purpose of the prototype and discusses its scope. The purpose of the User's Manual, which documents the steps to be followed for the demonstration, is addressed, and the chapter concludes with the format of the User's Manual.

6.2 The Scope of the Prototype

The Design Cycle of the SDLC consists of four main tasks as explained in *Figure 4.6*, namely a System Design task, an Analyze and Evaluate Design Alternatives task, and an Object Design task. The prototype concentrates on the System Design task which involves five steps. The first step was chosen to be supported by knowledge-based guidance, namely "Organize the system into sub-systems". The knowledge-based support which is given to this step of design was achieved in the Kappa-PC environment. In order to create the prototype, the following activities were required:

(i) The customization of Kappa-PC to contain a selection of rules to support the chosen step of the System Design task and the development of the user interface by means of the KAL language.

(ii) Choosing a suitable target system to be designed.

(iii) Establishing the analysis deliverables for the chosen target system and specifically verifying the Object Model.

(iv) Starting with the deliverables of point (iii), the design step to be supported is applied to the target system and the dialogue between the system designer and the rule base is demonstrated.

To illustrate the knowledge-based guidance, a target system was developed as a prototype. (The source code of the prototype is in *Appendix H.*) The problem statement of the target system, is as follows:
"*Design the software to support a computerized banking network including both human cashiers and automatic teller machines (ATMs) to be shared by a consortium of banks. Each bank provides its own computer to maintain its own accounts and process transactions against them. Cashier stations are owned by individual banks and communicate directly with their own bank's computers. Human cashiers enter account and transaction data. Automatic teller machines communicate with a central computer which clears transactions with the appropriate banks. An automatic teller machine accepts a cash card, interacts with the user, communicates with the central system to carry out the transaction, dispenses cash, and prints receipts. The system requires appropriate record-keeping and security provisions. The system must handle concurrent accesses to the same account correctly. The banks will provide their own software for their own computer; you are to design the software for the ATMs and the network. The cost of the shared system will be apportioned to the banks according to the number of customers with cash cards.*" (Rumbaugh et.al., 1991).

6.3 The Object-oriented Design User's Manual

The User's Manual is presented in *Appendix F* and serves as a guideline for activating and running the demonstration.

6.3.1 The Purpose of the Manual

The manual, presented in *Appendix F*, leads the user through a demonstration of limited scope to illustrate that it is possible to achieve knowledge-based support for the Design Cycle. The manual explains the use of the prototype stiffy and the correct reactions on the different questions posed when performing the relevant design step. The application software for this demonstration resides on the prototype stiffy which accompanies the dissertation.

6.3.2 The Format of the Manual

The User's Manual is a step-by-step guide for the user to help him in making the correct design choices, when executing the prototype, for purposes of the demonstration. The manual consists of instructions on how to activate the prototype; an example of a typical session follows and then the method to exit the prototype can be found.

6.4 Summary

In this chapter the purpose of the prototype is explained and the scope of the prototype is defined. By working with this prototype the designer is guided through the first step of the System Design task enabling him to obtain a feasible subsystem structure at an abstract level of the target system. A description of the demonstration can be found in *Appendix G* and the source code of the prototype is in *Appendix H.* A problem statement of the target system follows. The purpose and format of the User's Manual is clarified and outlined.

CHAPTER 7
Evaluation, Summary and Conclusions

CONTENTS

7.1 Introduction

A summary of the research results of the investigation is presented in this chapter. The original hypothesis and assumptions are validated in the light of these results, enabling conclusions to be drawn as presented here. The chapter concludes with proposed areas for further investigation.

7.2 Evaluation

The OMT methodology which was used during the investigation proved to be a continuous process since the three models which are created during analysis, namely the object model, the dynamic model and the functional model, are expanded and intensified during the Design Cycle. The same notation is used throughout all the SDLC cycles. The cycles are highly iterative. The Analysis, Design and Implementation Cycles may be repeated with more detail added in successive iterations. In this way incremental development is supported.

The knowledge-based environment which was used, namely Kappa-PC, provided sufficient and suitable support for the development of the prototype that was built for demonstration purposes.

The prototype itself supported the inexperienced system designer satisfactorily by leading him through a series of questions, forcing him into the correct school of thought and establishing the correct conclusion according to the answers. This demonstrates that it is possible to create an environment that can assist an inexperienced system designer to make design decisions.

7.3 Summary of Investigation

A hypothesis was postulated which stated that it is possible to create assistance for inexperienced system designers in a software development process, namely a knowledge-based support for object-oriented design. The relevant issues which have bearing on the investigation were identified, a motivation for the area of research was constructed and a method of investigation was established which guided the investigation. The assumptions which were made stated that the Analysis Cycle has been completed and the analysis deliverables are available. Keeping the established constraints in mind, a possible solution was proposed for dealing with the problems of constructing a good design in an object-oriented environment.

The design process was investigated. The software process model and the categories of design methods were explored, namely top-down structured design, data-driven design and object-oriented design. Object-oriented design was chosen because the parameters of the OISEE project prescribe it.

Another important aspect of the research is the support which knowledge-based systems may provide. A study was made of knowledge-based environments and one particular environment was chosen and used.

The steps of object-oriented design which may be supported by the knowledge base were identified and the conceptual model of a proposed solution was synthesized.

A significant amount of effort and time was required to master the sophisticated and technically advanced environment of Kappa-PC, described in Chapter 5. From the start, beginning with the key concepts of Kappa-PC, it was clear that the environment is more than merely a knowledge base. The different concepts in Kappa-PC were discussed, i.e. the KAL language, the end-user interface, the developer's interface and the external data sources interface together with the programming languages interface. Finally, the knowledge base's rule-based reasoning was addressed.

A prototype was built to serve as a demonstration of the concept. This required the customization of Kappa-PC to contain the relevant design rules and the user interface. Thereafter object-oriented design was performed on the chosen target system, with support from the Kappa-PC environment. Next a User's Manual was compiled to direct this demonstration which resides on the accompanying prototype stiffy.

7.4 Conclusions

The assumptions, namely that the Analysis Cycle is completed and that the analysis deliverables are available, were suitable assumptions and proved to be a sound point of departure for this investigation. The constraints proved to be valid ones because the PC environment, the Design Cycle of the revised spiral model and the parameters of the OISEE project guided the investigation on a focused and secure path.

The establishment of the relevant rules was feasible but the implementation of these rules in a specific environment proved to be more difficult.

The contribution of this research is that the original hypothesis that an inexperienced system designer, applying an object-oriented design methodology, may be supported by a knowledge-based environment was validated. This investigation also demonstrates that the technologies of SE, knowledge bases and software engineering environments may be combined to serve the development of quality software, thereby achieving higher levels of productivity among inexperienced system designers.

7.5 Areas for further Investigation

A number of areas were identified for further research. They will now be mentioned.

Quality assurance and verification of the Design Cycle are important issues and need more attention. The Quality Assurance Reference Model is being investigated by Ms D. Thornton, a member of OISEE project, in her Master's investigation.

Consistency and completeness of the Design Cycle must be established and must be verified before proceeding to implementation. These aspects need to be investigated.

Prototyping, especially during the Design Cycle, but also during the other cycles of the SDLC, extended and subjected to further investigation.

Aspects of re-usability during design, for example re-usable components (objects, classes or subsystems), design deliverables and documentation need further investigation.

Knowledge-based support for the other cycles of the SDLC is important and should be investigated.

The merging of the knowledge bases that provide support for the Feasibility Cycle, the Analysis Cycle, the Design Cycle and the Implementation Cycle of the SDLC, needs to be investigated. The result will be that the whole of the SDLC will be supported by knowledge-based guidance.

Literature References

Atkins M.C. and Brown A.W. 1991. *Principles of object-oriented systems* in Software Engineer's Reference Book edited by J. McDermid. Butterworth-Heinemann Ltd.

Boehm, B.W. August 1986. *A Spiral Model of Software Development and Enhancement*, in ACM SIGSOFT Software Engineering Notes, Vol.11, No.4.

Booch, G. February 1986. *Object-oriented Development*, in IEEE Transactions on Software Engineering, Vol.12, No.2, pp.211-221.

Booch, G. 1987. *Software Components with Ada: Structures, Tools, and Subsystems*. Menlo Park, California: Benjamin/Cummings Publishing Company, Inc.

Booch, G. 1991. *Object Oriented Design with Applications*. Redwood City, California: Benjamin/Cummings.

Bouzeghoub, G. 1985. *An Expert System for Database Design*, in Proceedings International Conference on VLDB.

Chabris Christopher F. 1988. *A Primer of Artificial Intelligence with sample programs in Turbo Pascal.* London, United Kingdom: Kogan Page Ltd.

Coad, P. and Yourdon, E. 1990. *Object-Oriented Analysis.* Englewood Cliffs, New Jersey: Yourdon Press.

Colter, M.A. 1982. *Evolution of The Structured Methodologies* in Advanced System Development/Feasibility Techniques, by Couger J.D., Colter M.A. and Knapp R.W. John Wiley & Sons.

Connor D. 1985. *Information System Specification and Design Road Map.* Englewood Cliffs, New Jersey: Prentice-Hall.

Cronk, R.N., Callahan, P.H. and Bernstein, L. September 1988. *Rule-Based Expert Systems for Network Management and Operations: An Introduction*, in IEEE Network, pp.7-21.

Dijkstra, E.W. January 1969. *Complexity controlled by hierarchical ordering of function and variability*, in Software Engineering, P.Naur and B.Randell, Eds. NATO.

Du Plessis, A.L. 1986. *A Software Engineering Environment for Real-time Systems*, PhD Thesis, University of South Africa.

Du Plessis, A.L. August 1992. *CAISE: The Opportunity and the Challenge.* Inaugural lecture University of South Africa.

Du Plessis, A.L. and Van der Walt, E. April 1992. *Modeling the Software Development Process*, in IFIP WG8.1 Working Conference on Information Systems Concepts: Improving the Understanding (ISCO2), Alexandria, Egypt.

Du Plessis, A.L. 1994. *Reuse in Information System Development*, Technical Document, Centre for Software Engineering (001/94), University of South Africa.

Forgy, C.L. 1979. *On the efficient implementation of production systems*, Ph.D.

Thesis, Carnegie-Mellon University.

Ford, N. 1991. *Expert Systems and Artificial Intelligence. An information manager's guide.* London:Library Association Publishing.

Forsyth R. 1989. *Expert Systems. Principles and Case Studies.* 2nd Ed. Chapman and Hall Ltd.

Harmon, P., Maus, R. and Morrissey, W. 1988. *Expert Systems: Tools and Applications.* New York: Wiley.

Harmsen, F. and Brinkkemper, S. 1993. *Computer Aided Method Engineering based on Existing Meta-CASE Technology*, in Proceedings of the Fourth Workshop on The next generation of CASE Tools, Memoranda Informatica 93-32.

Humphrey, W.S. 1989. *Managing the Software Process.* Addison-Wesley Publishing Company.

Holsapple, C.W. and Whinston, A.B. 1987. *Business Expert System.* Richard D.

Irwin, Inc., Homewood, Ill.

Ingalls, D. August 1981. *Design Principles behind Smalltalk.* Byte, Vol.6, No.8, p.286.

Jackson, M. 1975. *Principles of Program Design.* Orlando, FL: Academic Press.

Jackson, M. 1983. *System Development.* Englewood Cliffs, New Jersey: Prentice-Hall.

Kappa-PC Quick Start Manual. June 1992. Version 2, IntelliCorp, Inc., USA.

Kappa-PC User's Guide Manual. June 1992. Version 2, IntelliCorp, Inc., USA.

Klint, P. 1993. *A Meta-Environment for generating Programming Environments,* in ACM Transactions on Software Engineering and Methodology, Vol.2, No.2.

Liu, L. and Horowitz, E. October 1989. *A Formal Model for Software Project Management,* in IEEE Transactions on Software Engineering, Vol.15, No.10.

Loomis, M.E.S., Shah, A.V. and Rumbaugh, J.E. June 1987. *An object modeling technique for conceptual design*, in European Conference on Object-Oriented Programming, Paris, France, published as Lecture Notes in Computer Science, 276, Springer-Verlag.

Lucas P. and Van der Gaag, L. 1991. *Principles of Expert Systems.* Addison-Wesley Publishing Company.

Meyer, B. 1988. *Object-oriented Software Construction.* Englewood Cliffs, Hertfordshire: Prentice Hall.

Micallef, J. April/May 1988. *Encapsulation, Reusability, and Extensibility in Object-Oriented Programming Languages*, in Journal of Object-Oriented Programming, Vol.1, No.1, p.15.

Olle, T.W., Hagelstein, J., Macdonald, I.G., Rolland, C., Sol, H.G., Van Assche, F.J.M. and Verrijn-Stuart, A.A. 1988. *Information Systems Methodologies: A Framework for Understanding.* 2nd ed. 1991. England: Addison-Wesley.

Orr, K. 1971. *Structured Systems Development.* New York, NY: Yourdon Press.

Page-Jones, M. 1988. *The Practical Guide to Structured Systems Design.* 2nd ed. New Jersey: Prentice-Hall.

Parnas, D.L. December 1972. *On the Criteria to be used in Decomposing Systems into Modules*, in Communications of the ACM, Vol.15, No.2, pp.1053-1058.

Pocock, J.N. 1991. *Framework and tools for the integration of models*, Systematica technical Report M2K91.

Rolland, C. & Proix, C. 1986. *An Expert System Approach to Information System Design*, in Information Processing 86, editor H.J.Krugler. North-Holland:241-250.

Royce, W. W. 1970. *Managing the Development of Large Software Systems: Concepts and Techniques*, in Proceedings WESCON, pp.1-9.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. 1991. *Object-Oriented Modeling and Design.* New Jersey: Prentice-Hall.

Sage, A.P. and Palmer, J.D. 1990. *Software Systems Engineering*. New York: John Wiley & Sons.

Shinghal, R. 1992. *Formal Concepts in Artificial Intelligence.* Chapman & Hall.

Shlaer, S. and Mellor, S. J. 1988. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, New Jersey: Yourdon Press.

Sommerville, I. 1992. *Software Engineering.* Addison-Wesley Publishing Company Inc., 4th ed.

Stylianou, A.C., Madey G.R. & Smith, R.D. October 1992. *Selection Criteria for Expert System Shells: A Socio-Technical Framework*, in Communications of the ACM, Vol.35, No.10, pp.30-48.

Thomas, D. March 1989. *What's in an Object?*, in BYTE Vol.14, No.3, pp.231-240.

Turban, E. 1990. *Decision Support and Expert Systems: Management Support Systems*. New York: Macmillan Publishing Company, 2nd ed.

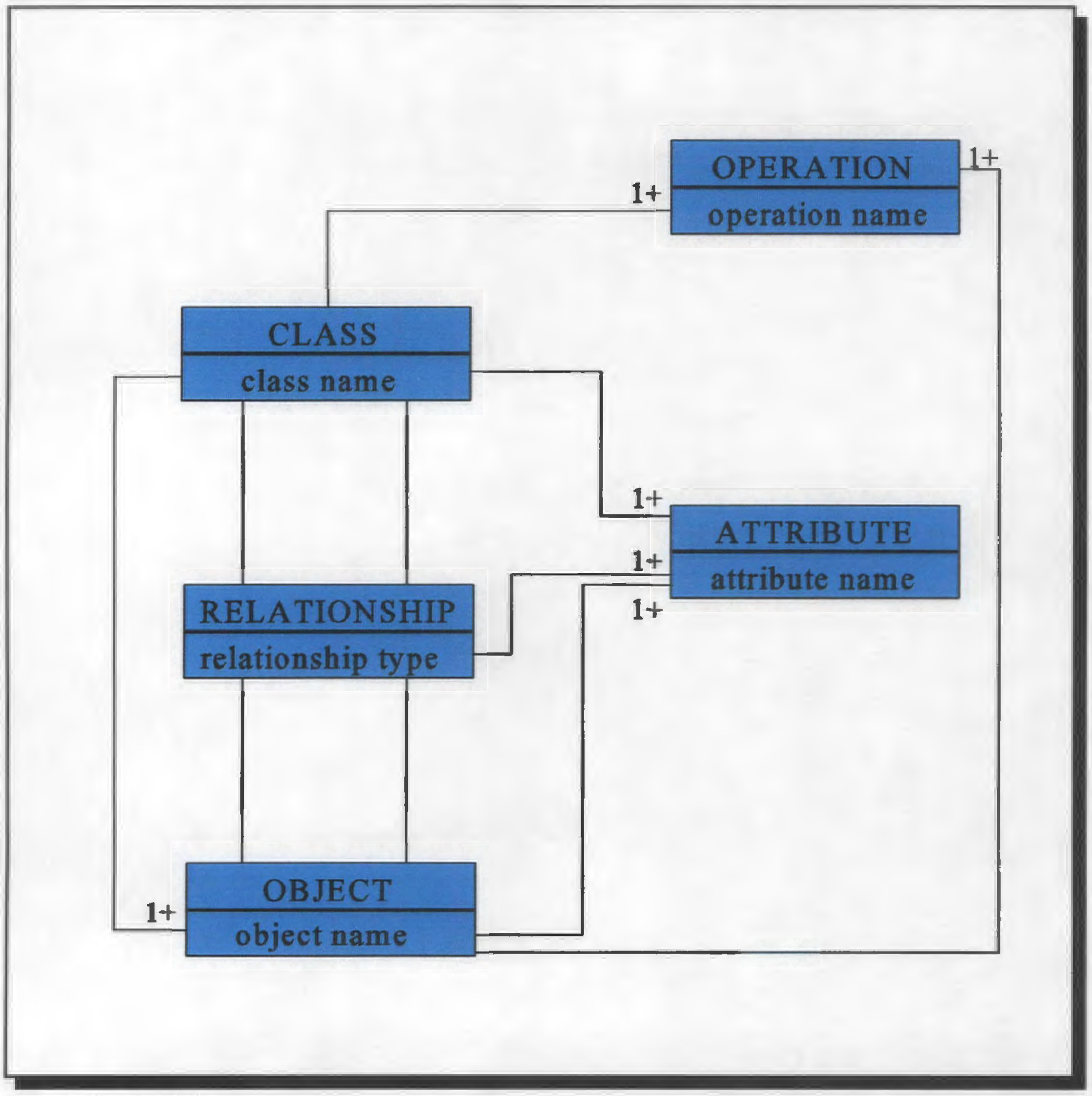
Van de Weg, R.L.W. & Engmann, R. April 1992. *A Framework and Method for Object-Oriented Information Systems Analysis and Design*, in IFIP WG8.1 Working Conference on Information Systems Concepts: Improving the Understanding, Alexandria, Egypt.

Van der Walt, E. 1994. *Software Project Management for Object-oriented development*, Msc Thesis, University of South Africa.

Waterman, D.A. 1986. *A Guide to Expert Systems.* Addison-Wesley Publishing Company.

EXHIBITS

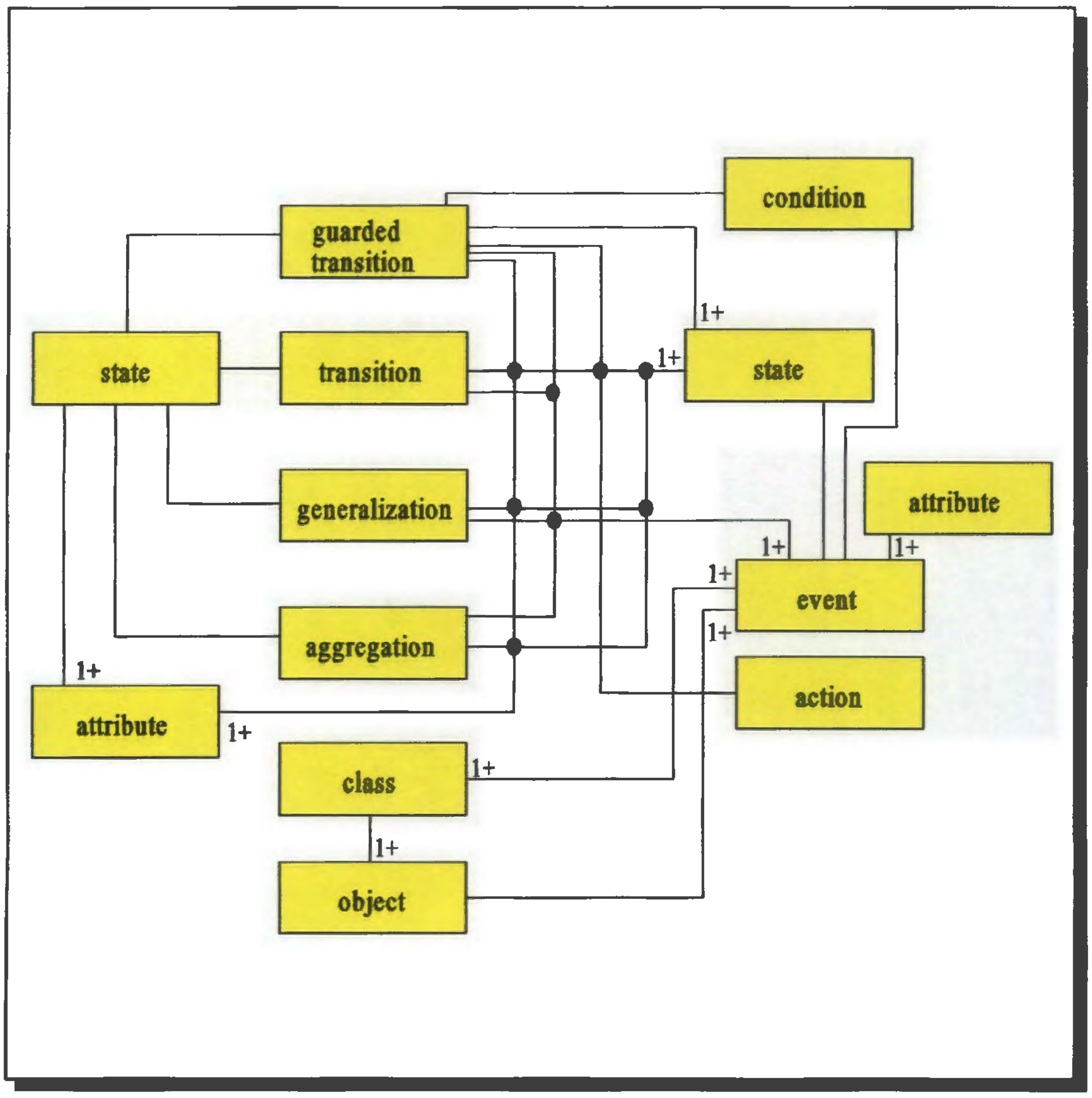
The object model presents the static structure of a system by showing the objects in the system, relationships between the objects, and the attributes and operations which characterize each class of objects. This model is compiled with primitives from *Table 4.1*.



Meta Object Model

Exhibit 4.1

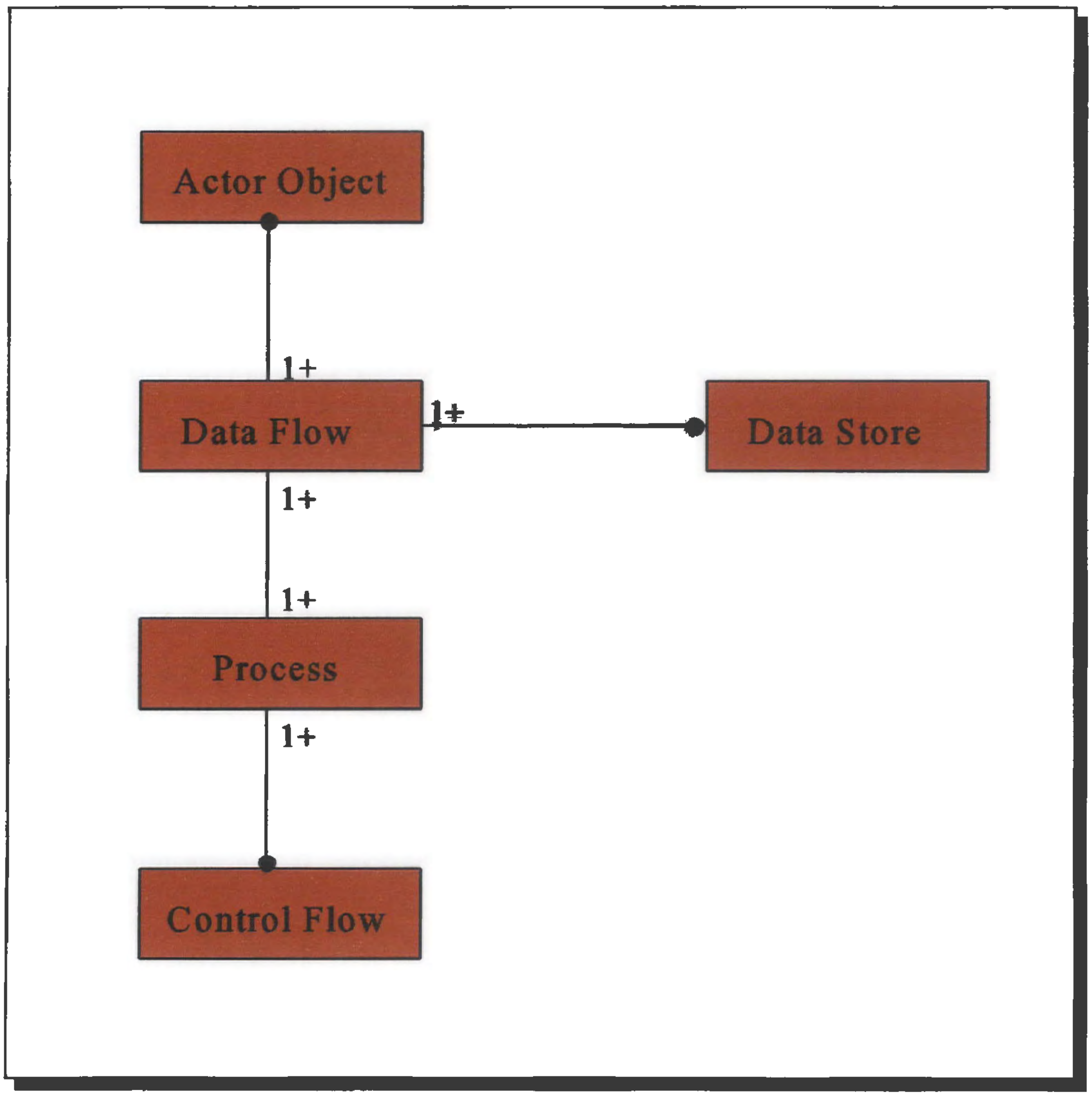
The dynamic model consists of multiple state diagrams, one state diagram for each CLASS with important dynamic behavior, and shows the pattern of activity for an entire system. This model is compiled with primitives from *Table 4.2*.



Meta Dynamic Model

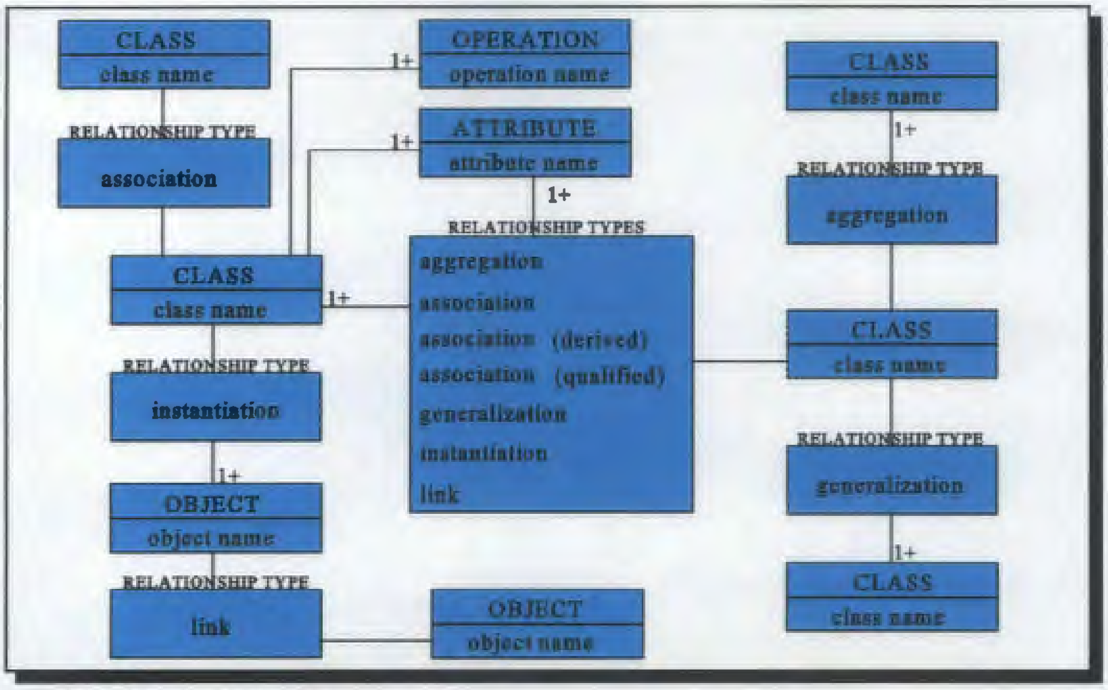
Exhibit 4.2

The functional model describes computations within a system and consists of multiple data flow diagrams which show the flow of values from external inputs, through operations and internal data stores, to external outputs. This model is compiled with primitives from *Table 4.3*.

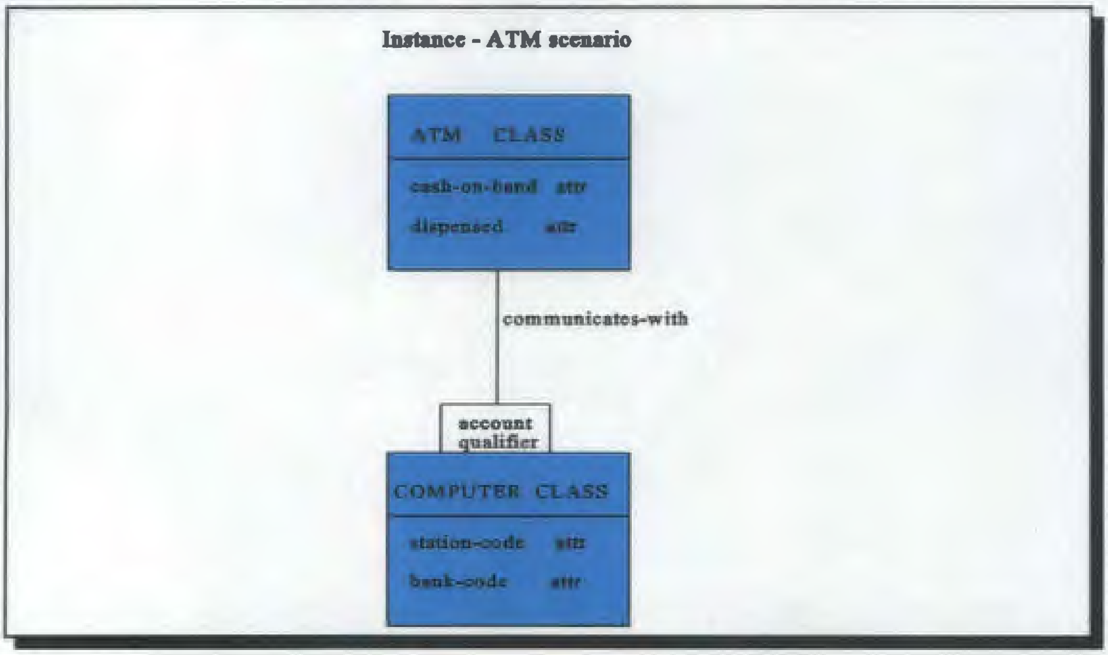


Meta Functional Model

Exhibit 4.3

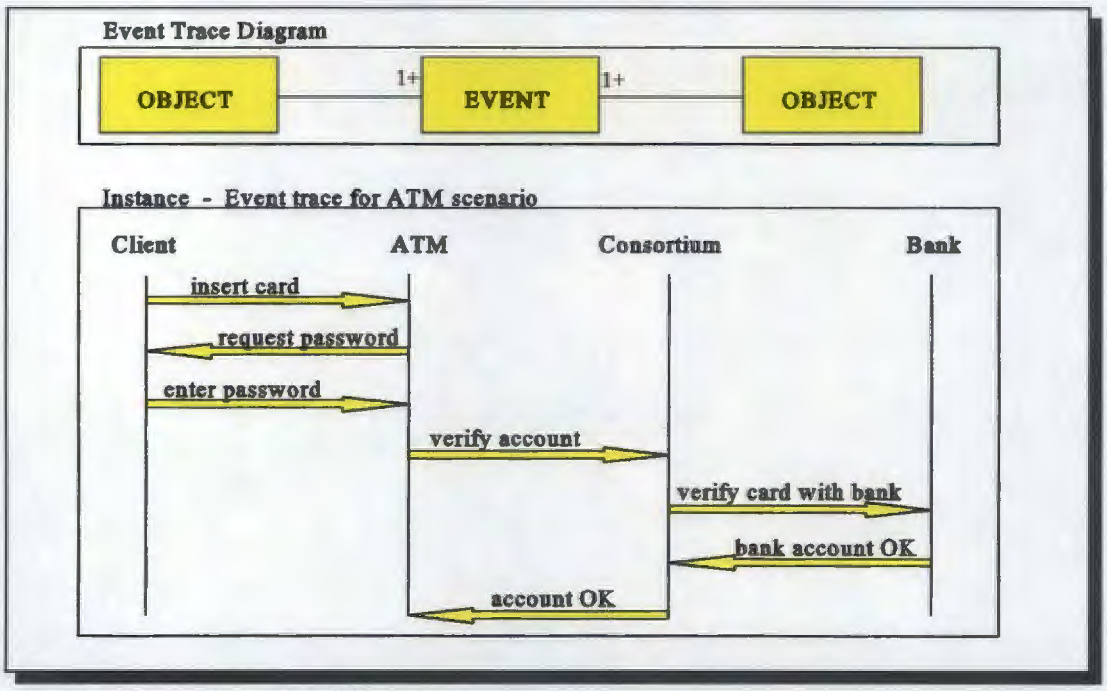


Meta Model for class/object diagrams

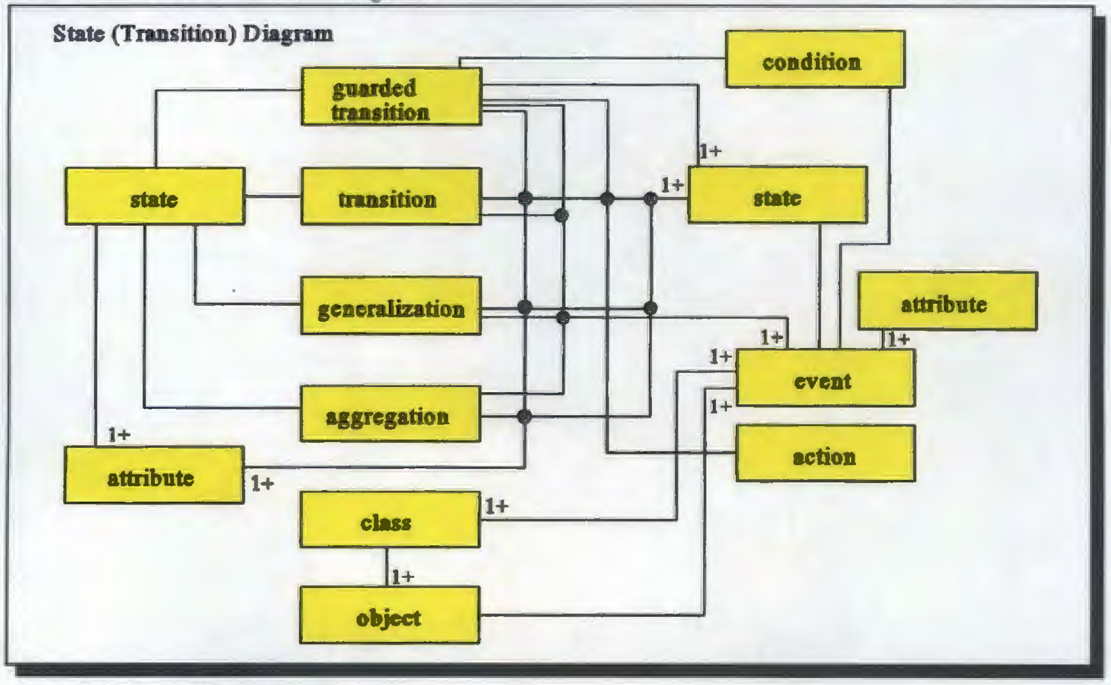


Instance of a Class Diagram

Exhibit 4.4

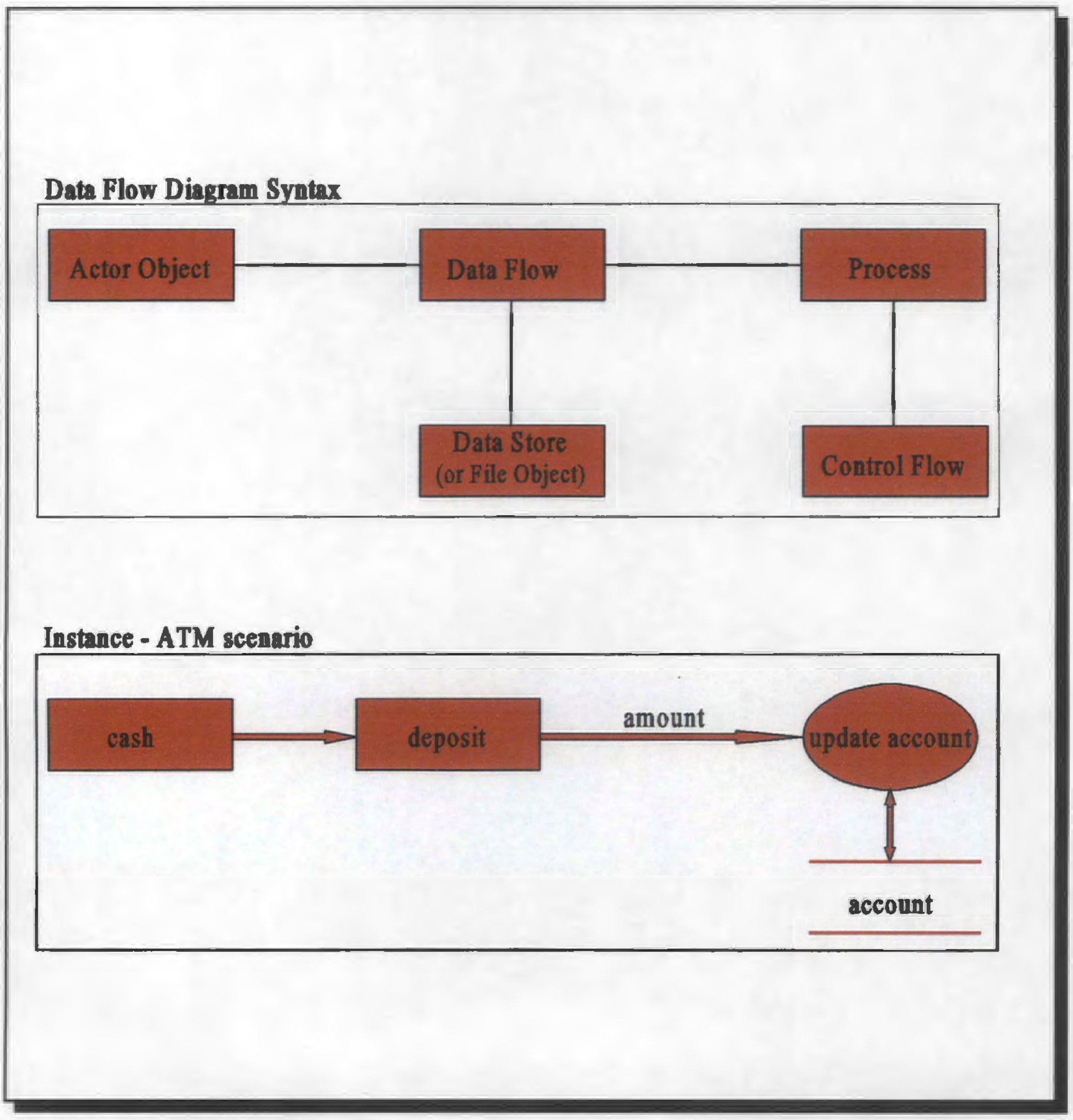


Meta Model for an Event Trace Diagram



Meta Model for a State Diagram

Exhibit 4.5



Meta Model for a Data Flow Diagram

Exhibit 4.6

APPENDIX A

Kappa-PC

The following is a subjective evaluation of Kappa-PC according to the author, and was done based on literature from the vendors, before empirical experience. For each category the criteria considered most important are double underlined and received a 3 if present, according to the above mentioned literature. The next important criteria are underlined and received a 2 if present. The rest of the criteria received a 1 if present (Stylianou *et al.*, 1992).

(i) ***End-User Interface Criteria***

- Saved Cases
- Explanation Facilities
    - Reasoning Path - How Graph 3
    - What - Paraphrases
    - Why - Relevances
- Documentation
- Tutorial
- Windows
    - Window Colors, Borders, Sizes
    - Menu System
        - Pop-Up Menus
        - Pull-Down Menus
    - Customizable Features 2
- Speech I/O
- Accepts Unknown as an Answer
- Context-Sensitive Help
- Display Manager
    - Graphic Results 2
    - Graphic Decision Tree
- Optimization

- Learning
- Mouse Support
- Natural Language Interface
- Sensitivity Analysis or Change Answers and Rerun

(ii) ***Developer Interface Criteria***

● Command Language/interpreter	1
● Documentation	
● Tutorial	
● Editing/Debugging Tools	
○ Rule/Working-Memory Browser	
○ Tracing	
○ Cross-Index Utility	
○ Incremental Compilation	
● Explanation Facility	
○ How (Reasoning Path)	3
○ What (Paraphrase)	
○ Why (Relevance)	
● Ability to Customize Explanations	3
● Graphics	2
● Mathematical Capabilities	
● Sample Knowledge Bases	
● Code Generator	
● Windows	
○ Window Colors, Borders, Sizes	1
○ Menu System	
▫ Pop-Up Menus	1
▫ Pull-Down Menus	1
○ Customizable Features	2
● Rapid Prototyping	3
● Open Architecture	

- Batch Processing Facilities
- Novice/Expert Modes
- String Handling

(iii) ***System Interface Criteria***

- Hardware
  - Portability
  - Support for Microcomputers 2
  - Compatibility
  - Multi-processor Support
  - Multi-user Support
  - Access to Special Hardware
- Implementation Language
  - Portability
  - Embeddability 3
  - Compatibility
- Copy Protection
- Batch Processing
- Real-Time Processing
- Network Support

(iv) ***Inference Engine Criteria***

- Reasoning Mode
  - Forward Chaining 2
  - Backward Chaining 3
  - Bi-Directional Inferencing
  - Non-monotonic Reasoning
- Truth Maintenance System
- Search Strategy
  - Breadth First 2
  - Depth First 2

- o Branch-And-Bound
- o Generate And Test
- o Best First 2
- o Hill Climbing

- ● Find All Answers
- ● Find Only One Answer
- ● Conflict Resolution
  - o Rule-Assigned Priority 2
  - o Specificity
  - o Recency
- ● Certainty Measurement
  - o Bayes Theorem
  - o Certainty Factor Model
  - o Dempster-Shafer Theory
  - o Fuzzy Set Theory
  - o Inheritance
  - o Certainty Threshold
- ● Blackboard
- ● Recursion
- ● Iteration
- ● Fuzzy Sets
- ● Reliability

(v) Knowledge Base Criteria

- ● Representation Technique
  - o Rules 2
  - o Partitioned Rule Sets
  - o Meta-rules
  - o Decision Tables
  - o Frames
  - o Scripts/Schemata

○ Semantic Networks	
○ Formal Logic	
● Induction	
● Inheritance	2
● Knowledge Engineering Sub-system	
● Multiple Instance	
● Demons	1
● Case Management	
● Capacity	

(vi) ***Data Interface Criteria***

● Access to 3GL and 4GL	
● Linkage to Databases	3
● Access to Underlying Language	2
● Linkage to Special Purpose Software	
○ Linkage to Transaction Processing Environments	2
○ Access to Lotus, DOS, etc.	2

(vii) ***Cost-Related Criteria***

- Upgrades
- Required Software/Hardware
- Conversion
- Personnel
- Vendor Technical Support
- Training Programs
- Installation
- Run-Time Licence
- Consulting Fees

(viii) Vendor-Related Criteria

- Maintenance
- Technical Support
- Training Courses
- Professional Application Development Services
- Product/Vendor Maturity
- Commitment to Product
- Upgrade Path

Total : 56

APPENDIX B

Leonardo

The following is a subjective evaluation of Leonardo according to the author, and was done based on literature from the vendors, before empirical experience. For each category the criteria considered most important are double underlined and received a 3 if present, according to the above mentioned literature. The next important criteria are underlined and received a 2 if present. The rest of the criteria received a 1 if present (Stylianou *et al.*, 1992).

(i) ***End-User Interface Criteria***

- Saved Cases
- Explanation Facilities
    - Reasoning Path - How Graph
    - What - Paraphrases
    - Why - Relevances 3
- Documentation
- Tutorial
- Windows
    - Window Colors, Borders, Sizes
    - Menu System
        - Pop-Up Menus
        - Pull-Down Menus
    - Customizable Features 2
- Speech I/O
- Accepts Unknown as an Answer
- Context-Sensitive Help
- Display Manager
    - Graphic Results 2
    - Graphic Decision Tree
- Optimization

- Learning
- Mouse Support
- Natural Language Interface
- Sensitivity Analysis or Change Answers and Rerun

(ii) ***Developer Interface Criteria***

- Command Language/interpreter
- Documentation 3
- Tutorial
- Editing/Debugging Tools
  - Rule/Working-Memory Browser 2
  - Tracing 2
  - Cross-Index Utility
  - Incremental Compilation
- Explanation Facility
  - How (Reasoning Path) 3
  - What (Paraphrase)
  - Why (Relevance) 3
- Ability to Customize Explanations 3
- Graphics
- Mathematical Capabilities 2
- Sample Knowledge Bases
- Code Generator
- Windows
  - Window Colors, Borders, Sizes
  - Menu System
    - Pop-Up Menus
    - Pull-Down Menus
  - Customizable Features
- Rapid Prototyping
- Open Architecture 2

- Batch Processing Facilities
- Novice/Expert Modes
- String Handling

(iii) ***System Interface Criteria***

- Hardware
  - Portability
  - Support for Microcomputers
  - Compatibility
  - Multi-processor Support
  - Multi-user Support
  - Access to Special Hardware
- Implementation Language
  - Portability
  - Embeddability
  - Compatibility
- Copy Protection
- Batch Processing
- Real-Time Processing
- Network Support

(iv) ***Inference Engine Criteria***

- Reasoning Mode
  - Forward Chaining 2
  - Backward Chaining 3
  - Bi-Directional Inferencing 2
  - Non-monotonic Reasoning
- Truth Maintenance System
- Search Strategy
  - Breadth First 2
  - Depth First 2

o Branch-And-Bound	
o Generate And Test	
o <u>Best First</u>	
o Hill Climbing	
• <u>Find All Answers</u>	
• Find Only One Answer	
• Conflict Resolution	
o <u>Rule-Assigned Priority</u>	
o Specificity	
o Recency	
• Certainty Measurement	
o Bayes Theorem	1
o <u>Certainty Factor Model</u>	2
o Dempster-Shafer Theory	
o Fuzzy Set Theory	
o <u>Inheritance</u>	
o Certainty Threshold	
• Blackboard	
• <u>Recursion</u>	
• <u>Iteration</u>	
• Fuzzy Sets	
• Reliability	

(v) ***Knowledge Base Criteria***

• Representation Technique	
o <u>Rules</u>	2
o <u>Partitioned Rule Sets</u>	2
o <u>Meta-rules</u>	
o Decision Tables	
o <u>Frames</u>	2
o Scripts/Schemata	

○ Semantic Networks	
○ Formal Logic	
● Induction	
● <u>Inheritance</u>	
● <u>Knowledge Engineering Sub-system</u>	
● <u>Multiple Instance</u>	
● Demons	1
● Case Management	
● Capacity	

(vi) Data Interface Criteria

● Access to 3GL and 4GL	
● <u>Linkage to Databases</u>	3
● <u>Access to Underlying Language</u>	
● <u>Linkage to Special Purpose Software</u>	
○ Linkage to Transaction Processing Environments	
○ Access to Lotus, DOS, etc.	2

(vii) Cost-Related Criteria

- Upgrades
- Required Software/Hardware
- Conversion
- Personnel
- Vendor Technical Support
- Training Programs
- Installation
- Run-Time Licence
- Consulting Fees

(viii) ***Vendor-Related Criteria***

- Maintenance
- Technical Support
- Training Courses
- Professional Application Development Services
- Product/Vendor Maturity
- Commitment to Product
- Upgrade Path

Total : 53

APPENDIX C

Nexpert Object

The following is a subjective evaluation of Nexpert Object according to the author, and was done based on literature from the vendors, before empirical experience. For each category the criteria considered most important are double underlined and received a 3 if present, according to the above mentioned literature. The next important criteria are underlined and received a 2 if present. The rest of the criteria received a 1 if present (Stylianou *et al.*, 1992).

(i) ***End-User Interface Criteria***

- Saved Cases
- Explanation Facilities
    - Reasoning Path - How Graph
    - What - Paraphrases
    - Why - Relevances
- Documentation
- Tutorial
- Windows
    - Window Colors, Borders, Sizes
    - Menu System
        - Pop-Up Menus
        - Pull-Down Menus
    - Customizable Features
- Speech I/O
- Accepts Unknown as an Answer
- Context-Sensitive Help
- Display Manager
    - Graphic Results
    - Graphic Decision Tree
- Optimization

- Learning
- Mouse Support
- Natural Language Interface
- Sensitivity Analysis or Change Answers and Rerun

(ii) ***Developer Interface Criteria***

- Command Language/interpreter
- Documentation
- Tutorial
- Editing/Debugging Tools
    - Rule/Working-Memory Browser 2
    - Tracing
    - Cross-Index Utility
    - Incremental Compilation
- Explanation Facility
    - How (Reasoning Path)
    - What (Paraphrase)
    - Why (Relevance)
- Ability to Customize Explanations
- Graphics
- Mathematical Capabilities
- Sample Knowledge Bases
- Code Generator
- Windows
    - Window Colors, Borders, Sizes
    - Menu System
        - Pop-Up Menus
        - Pull-Down Menus
    - Customizable Features
- Rapid Prototyping
- Open Architecture

- Batch Processing Facilities
- Novice/Expert Modes
- <u>String Handling</u>

(iii) ***System Interface Criteria***

- Hardware
  - <u>Portability</u>
  - <u>Support for Microcomputers</u>
  - <u>Compatibility</u>
  - Multi-processor Support
  - Multi-user Support
  - Access to Special Hardware
- <u>Implementation Language</u>
  - <u>Portability</u>
  - <u>Embeddability</u>
  - <u>Compatibility</u>
- <u>Copy Protection</u>
- Batch Processing
- <u>Real-Time Processing</u>
- Network Support

(iv) ***Inference Engine Criteria***

- <u>Reasoning Mode</u>
  - <u>Forward Chaining</u> 2
  - <u>Backward Chaining</u> 3
  - <u>Bi-Directional Inferencing</u>
  - <u>Non-monotonic Reasoning</u>
- <u>Truth Maintenance System</u>
- Search Strategy
  - <u>Breadth First</u>
  - <u>Depth First</u>

- Branch-And-Bound
- Generate And Test
- <u>Best First</u>
- Hill Climbing

- <u>Find All Answers</u>
- Find Only One Answer
- Conflict Resolution
  - <u>Rule-Assigned Priority</u>
  - Specificity
  - Recency
- Certainty Measurement
  - Bayes Theorem
  - <u>Certainty Factor Model</u>
  - Dempster-Shafer Theory
  - Fuzzy Set Theory
  - <u>Inheritance</u>
  - Certainty Threshold
- Blackboard
- <u>Recursion</u>
- <u>Iteration</u>
- Fuzzy Sets
- Reliability

(v) ***Knowledge Base Criteria***

- Representation Technique
  - <u>Rules</u> 2
  - <u>Partitioned Rule Sets</u>
  - <u>Meta-rules</u>
  - Decision Tables
  - <u>Frames</u>
  - Scripts/Schemata

    - Semantic Networks
    - Formal Logic
- Induction
- Inheritance
- Knowledge Engineering Sub-system
- Multiple Instance
- Demons
- Case Management
- Capacity

(vi) ***Data Interface Criteria***

- Access to 3GL and 4GL 1
- Linkage to Databases 3
- Access to Underlying Language
- Linkage to Special Purpose Software
    - Linkage to Transaction Processing Environments
    - Access to Lotus, DOS, etc. 2

(vii) ***Cost-Related Criteria***

- Upgrades
- Required Software/Hardware
- Conversion
- Personnel
- Vendor Technical Support
- Training Programs
- Installation
- Run-Time Licence
- Consulting Fees

(viii) Vendor-Related Criteria

- Maintenance
- Technical Support
- Training Courses
- Professional Application Development Services
- Product/Vendor Maturity
- Commitment to Product
- Upgrade Path

Total : 15

APPENDIX D

ART-IM

The following is a subjective evaluation of ART-IM according to the author, and was done based on literature from the vendors, before empirical experience. For each category the criteria considered most important are double underlined and received a 3 if present, according to the above mentioned literature. The next important criteria are underlined and received a 2 if present. The rest of the criteria received a 1 if present (Stylianou *et al.*, 1992).

(i) ***End-User Interface Criteria***

- Saved Cases
- Explanation Facilities
    - Reasoning Path - How Graph
    - What - Paraphrases
    - Why - Relevances
- Documentation
- Tutorial
- Windows
    - Window Colors, Borders, Sizes
    - Menu System
        - Pop-Up Menus
        - Pull-Down Menus
    - Customizable Features
- Speech I/O
- Accepts Unknown as an Answer
- Context-Sensitive Help
- Display Manager
    - Graphic Results
    - Graphic Decision Tree
- Optimization

- Learning
- Mouse Support
- Natural Language Interface
- Sensitivity Analysis or Change Answers and Rerun

(ii) ***Developer Interface Criteria***

- Command Language/interpreter
- Documentation
- Tutorial
- Editing/Debugging Tools
    - Rule/Working-Memory Browser
    - Tracing 2
    - Cross-Index Utility
    - Incremental Compilation 2
- Explanation Facility
    - How (Reasoning Path)
    - What (Paraphrase)
    - Why (Relevance) 1
- Ability to Customize Explanations
- Graphics
- Mathematical Capabilities
- Sample Knowledge Bases
- Code Generator
- Windows
    - Window Colors, Borders, Sizes
    - Menu System
        - Pop-Up Menus
        - Pull-Down Menus
    - Customizable Features
- Rapid Prototyping
- Open Architecture

	● Batch Processing Facilities	
	● Novice/Expert Modes	
	● String Handling	
(iii)	*System Interface Criteria*	
	● Hardware	
	○ Portability	2
	○ Support for Microcomputers	
	○ Compatibility	
	○ Multi-processor Support	
	○ Multi-user Support	
	○ Access to Special Hardware	
	● Implementation Language	
	○ Portability	
	○ Embeddability	
	○ Compatibility	
	● Copy Protection	
	● Batch Processing	
	● Real-Time Processing	2
	● Network Support	
(iv)	*Inference Engine Criteria*	
	● Reasoning Mode	
	○ Forward Chaining	2
	○ Backward Chaining	
	○ Bi-Directional Inferencing	
	○ Non-monotonic Reasoning	
	● Truth Maintenance System	2
	● Search Strategy	
	○ Breadth First	2
	○ Depth First	2

- ○ Branch-And-Bound
- ○ Generate And Test
- ○ Best First   2
- ○ Hill Climbing

- ● Find All Answers
- ● Find Only One Answer
- ● Conflict Resolution
  - ○ Rule-Assigned Priority
  - ○ Specificity
  - ○ Recency
- ● Certainty Measurement
  - ○ Bayes Theorem
  - ○ Certainty Factor Model
  - ○ Dempster-Shafer Theory
  - ○ Fuzzy Set Theory
  - ○ Inheritance
  - ○ Certainty Threshold
- ● Blackboard
- ● Recursion
- ● Iteration
- ● Fuzzy Sets
- ● Reliability

(v) ***Knowledge Base Criteria***

- ● Representation Technique
  - ○ Rules   2
  - ○ Partitioned Rule Sets
  - ○ Meta-rules
  - ○ Decision Tables
  - ○ Frames   2
  - ○ Scripts/Schemata

- Semantic Networks
- Formal Logic

● Induction	
● <u>Inheritance</u>	
● <u>Knowledge Engineering Sub-system</u>	
● <u>Multiple Instance</u>	
● Demons	
● Case Management	
● Capacity	

(vi) ***Data Interface Criteria***

● Access to 3GL and 4GL	
● <u>Linkage to Databases</u>	3
● <u>Access to Underlying Language</u>	
● <u>Linkage to Special Purpose Software</u>	
o Linkage to Transaction Processing Environments	
o Access to Lotus, DOS, etc.	2

(vii) ***Cost-Related Criteria***

- Upgrades
- Required Software/Hardware
- Conversion
- Personnel
- Vendor Technical Support
- Training Programs
- Installation
- Run-Time Licence
- Consulting Fees

(viii) ***Vendor-Related Criteria***

- Maintenance
- Technical Support
- Training Courses
- Professional Application Development Services
- Product/Vendor Maturity
- Commitment to Product
- Upgrade Path

Total : 28

APPENDIX E

EXSYS Professional

The following is a subjective evaluation of EXSYS Professional according to the author, and was done based on literature from the vendors, before empirical experience. For each category the criteria considered most important are double underlined and received a 3 if present, according to the above mentioned literature. The next important criteria are underlined and received a 2 if present. The rest of the criteria received a 1 if present (Stylianou *et al.*, 1992).

(i) ***End-User Interface Criteria***

- Saved Cases
- Explanation Facilities
  - Reasoning Path - How Graph
  - What - Paraphrases
  - Why - Relevances
- Documentation
- Tutorial
- Windows
  - Window Colors, Borders, Sizes
  - Menu System
    - Pop-Up Menus
    - Pull-Down Menus
  - Customizable Features
- Speech I/O
- Accepts Unknown as an Answer
- Context-Sensitive Help
- Display Manager
  - Graphic Results
  - Graphic Decision Tree
- Optimization

- Learning
- Mouse Support
- Natural Language Interface
- Sensitivity Analysis or Change Answers and Rerun

(ii) ***Developer Interface Criteria***

- Command Language/interpreter
- Documentation
- Tutorial
- Editing/Debugging Tools
  - Rule/Working-Memory Browser
  - Tracing
  - Cross-Index Utility
  - Incremental Compilation
- Explanation Facility
  - How (Reasoning Path)
  - What (Paraphrase)
  - Why (Relevance)
- Ability to Customize Explanations
- Graphics 2
- Mathematical Capabilities 2
- Sample Knowledge Bases
- Code Generator
- Windows
  - Window Colors, Borders, Sizes
  - Menu System
    - Pop-Up Menus
    - Pull-Down Menus
  - Customizable Features 2
- Rapid Prototyping
- Open Architecture

- Batch Processing Facilities
- Novice/Expert Modes
- String Handling

(iii) ***System Interface Criteria***

- Hardware
  - Portability
  - Support for Microcomputers
  - Compatibility
  - Multi-processor Support
  - Multi-user Support
  - Access to Special Hardware
- Implementation Language
  - Portability
  - Embeddability
  - Compatibility
- Copy Protection
- Batch Processing
- Real-Time Processing
- Network Support

(iv) ***Inference Engine Criteria***

- Reasoning Mode
  - Forward Chaining 2
  - Backward Chaining 3
  - Bi-Directional Inferencing
  - Non-monotonic Reasoning
- Truth Maintenance System
- Search Strategy
  - Breadth First
  - Depth First

o Branch-And-Bound	
o Generate And Test	
o <u>Best First</u>	
o Hill Climbing	
● <u>Find All Answers</u>	
● Find Only One Answer	
● Conflict Resolution	
o <u>Rule-Assigned Priority</u>	
o Specificity	
o Recency	
● Certainty Measurement	
o Bayes Theorem	
o <u>Certainty Factor Model</u>	2
o Dempster-Shafer Theory	
o Fuzzy Set Theory	
o <u>Inheritance</u>	
o Certainty Threshold	
● Blackboard	
● <u>Recursion</u>	
● <u>Iteration</u>	
● Fuzzy Sets	
● Reliability	

(v) ***Knowledge Base Criteria***

● Representation Technique	
o <u>Rules</u>	2
o <u>Partitioned Rule Sets</u>	2
o <u>Meta-rules</u>	
o Decision Tables	
o <u>Frames</u>	2
o Scripts/Schemata	

  - ○ Semantic Networks
  - ○ Formal Logic
- Induction
- Inheritance 2
- Knowledge Engineering Sub-system
- Multiple Instance
- Demons
- Case Management
- Capacity

(vi) ***Data Interface Criteria***

- Access to 3GL and 4GL
- Linkage to Databases 3
- Access to Underlying Language
- Linkage to Special Purpose Software
  - ○ Linkage to Transaction Processing Environments
  - ○ Access to Lotus, DOS, etc. 2

(vii) ***Cost-Related Criteria***

- Upgrades
- Required Software/Hardware
- Conversion
- Personnel
- Vendor Technical Support
- Training Programs
- Installation
- Run-Time Licence
- Consulting Fees

(viii) Vendor-Related Criteria

- Maintenance
- Technical Support
- Training Courses
- Professional Application Development Services
- Product/Vendor Maturity
- Commitment to Product
- Upgrade Path

Total : 26

APPENDIX F

User's Manual

This prototype was developed on Kappa-PC 2.0.

(The source code of the prototype is in *Appendix H.*)

When working with the prototype stiffy that accompanies the dissertation:

1. Activate Windows.
2. Activate Kappa-PC 2.0 and proceed with Step 3.

When working on the PC in Room 8-86 Theo van Wijk building, UNISA:

1. Click with left Mouse button twice on the ***Compilers*** icon.
2. Click with left Mouse button twice on the ***Kappa-PC 2.0*** icon and *wait.*

3. Click with left Mouse button once on ***File*** in KAPPA (untitled) window.
4. Click with left Mouse button once on ***Open.***
5. Click with left Mouse button once on ***design.kal*** (when working on the PC in Room 8-86) or type ***b:design.kal*** in the **File Name:** position (when working with the stiffy).
6. Click with left Mouse button once on ***OK*** and *wait.*
7. Click with left Mouse button once on the ***Session*** icon in the KAPPA (untitled) window.
8. Click with left Mouse button once on ***SESSION*** and then click on ***OK.*** (The first window of the Design Cycle - *Figure G.3* in *Appendix G* - will appear.)
9. Click with left Mouse button once on ***Project4.***
10. Click with left Mouse button once on ***Design Project.***

(Refer to *Figure G.4* in *Appendix G.*)

11. Click with left Mouse button once on ***System Design.***
    (Refer to *Figure G.6* in *Appendix G.*)
12. Click with left Mouse button once on ***Step 1.***
    (Refer to *Figure G.7* in *Appendix G.*)
    (The three questions applicable to Step 1, relative to this prototype, begin.)
13. Click with left Mouse button once on ***OK.***
    (Refer to *Figure G.8* in *Appendix G.*)
14. Click with left Mouse button once on ***MainMiniComputer*** for *Transaction*-class.
15. Click with left Mouse button once on ***MainMiniComputer*** for *EntryStation*-class.
16. Click with left Mouse button once on ***MainMiniComputer*** for *CashierTransaction*-class.
17. Click with left Mouse button once on ***SpecialComponent*** for *RemoteTransaction*-class.
18. Click with left Mouse button once on ***MainMiniComputer*** for *Update*-class.
19. Click with left Mouse button once on ***SpecialComponent*** for *ATM*-class.
20. Click with left Mouse button once on ***MainMiniComputer*** for *CashierStation*-class.
21. Click with left Mouse button once on ***MainMiniComputer*** for *Cashier*-class.
22. Click with left Mouse button once on ***MainMiniComputer*** for *CardAuthorization*-class.
23. Click with left Mouse button once on ***MainMiniComputer*** for *Customer*-

class.

24. Click with left Mouse button once on ***MainMiniComputer*** for *Consortium*-class.
25. Click with left Mouse button once on ***MainMiniComputer*** for *Bank*-class.
26. Click with left Mouse button once on ***MainMiniComputer*** for *Account*-class.
27. Click with left Mouse button once on ***SpecialComponent*** for *CashCard*-class.

    (This concludes Question 1 of Step 1. Question 2 of Step 1 starts now.)
    (Refer to *Figure G.9* in *Appendix G.*)
28. Click with left Mouse button once on ***OK***.

    (Refer to *Figure G.10* in *Appendix G.*)
29. Click with left Mouse button once on ***CentralMiniMainframe*** for *Transaction*-class.
30. Click with left Mouse button once on ***CentralMiniMainframe*** for *EntryStation*-class.
31. Click with left Mouse button once on ***CentralMiniMainframe*** for *CashierTransaction*-class.
32. Click with left Mouse button once on ***ExternalMiniMainframe*** for *RemoteTransaction*-class.
33. Click with left Mouse button once on ***CentralMiniMainframe*** for *Update*-class.
34. Click with left Mouse button once on ***ExternalMiniMainframe*** for *ATM*-class.
35. Click with left Mouse button once on ***CentralMiniMainframe*** for

CashierStation-class.

36. Click with left Mouse button once on ***CentralMiniMainframe*** for *Cashier*-class.
37. Click with left Mouse button once on ***CentralMiniMainframe*** for *CardAuthorization*-class.
38. Click with left Mouse button once on ***CentralMiniMainframe*** for *Customer*-class.
39. Click with left Mouse button once on ***ExternalMiniMainframe*** for *Consortium*-class.
40. Click with left Mouse button once on ***CentralMiniMainframe*** for *Bank*-class.
41. Click with left Mouse button once on ***CentralMiniMainframe*** for *Account*-class.
42. Click with left Mouse button once on ***ExternalMiniMainframe*** for *CashCard*-class.

    (This concludes Question 2 of Step 1. Question 3 of Step 1 starts now.) (Refer to *Figure G.11* in *Appendix G.*)
43. Click with left Mouse button once on ***OK***.

    (Refer to *Figure G.12* in *Appendix G.*)
44. Click with left Mouse button once on ***PerformArithmetic*** for *Transaction*-class.
45. Click with left Mouse button once on ***UserInterface*** for *EntryStation*-class.
46. Click with left Mouse button once on ***InputProcessing*** for *CashierTransaction*-class.
47. Click with left Mouse button once on ***Other*** for *RemoteTransaction*-

class. (Refer to *Figure G.13* in *Appendix G.*)

48. Move with down-arrow-key to third position and change ***Other*** to ***ATM.***
49. Click with left Mouse button once on ***OK.***
50. Click with left Mouse button once on ***PerformArithmetic*** for *Update*-class.
51. Click with left Mouse button once on ***Other*** for *ATM*-class.
52. Move with down-arrow-key to third position and change ***Other*** to ***ATM.***
53. Click with left Mouse button once on ***OK.***
54. Click with left Mouse button once on ***UserInterface*** for *CashierStation*-class.
55. Click with left Mouse button once on ***UserInterface*** for *Cashier*-class.
56. Click with left Mouse button once on ***PerformArithmetic*** for *CardAuthorization*-class.
57. Click with left Mouse button once on ***UserInterface*** for *Customer*-class.
58. Click with left Mouse button once on ***PerformArithmetic*** for *Consortium*-class.
59. Click with left Mouse button once on ***PerformArithmetic*** for *Bank*-class.
60. Click with left Mouse button once on ***PerformArithmetic*** for *Account*-class.
61. Click with left Mouse button once on ***Other*** for *CashCard*-class.
62. Move with down-arrow-key to third position and change ***Other*** to ***ATM.***
63. Click with left Mouse button once on ***OK*** and *wait.*

At this stage, because of the physical-location question (Question 2), there are two subsystems without names. They must now be named. (Refer to *Figure G.14* in *Appendix G.*)

64. Click with left Mouse button once on ***left white block***, type

ConsortiumComputer and ***press Enter.***

65. Click with left Mouse button once on ***right white block***, type ***BankComputers*** and ***press Enter.***
66. Click with left Mouse button once on ***Proceed.***
    (Refer to *Figure G.15* in *Appendix G.*)
67. Click with left Mouse button once on ***down-arrow of right white block.***
68. Click with left Mouse button once on ***ATMSubSystem*** in the overlay box on top of the big green box.

The Expert System infers that there are three major subsystems after the first iteration of Step 1 for the System Design task, namely "Breaking a system into subsystems". The three subsystems are shown as three big boxes in *Figure G.15.*

69. Click with left Mouse button once on ***Proceed.***
    (Refer to *Figure G.16* in *Appendix G.*)
70. Click with left Mouse button once on ***The End.***
71. Close all ***Kappa*** windows.
72. Close ***KAPPA (untitled)*** window.
73. Save changes? *NO.*

APPENDIX G

Description of Demonstration

Automated Teller Machine Example

The problem statement in Chapter 6 for an automated teller machine (ATM) network, shown in *Figure G.1*, serves as an example for the target system. This ATM problem is used for purposes of the prototype. The source code for the prototype is in *Appendix H*.

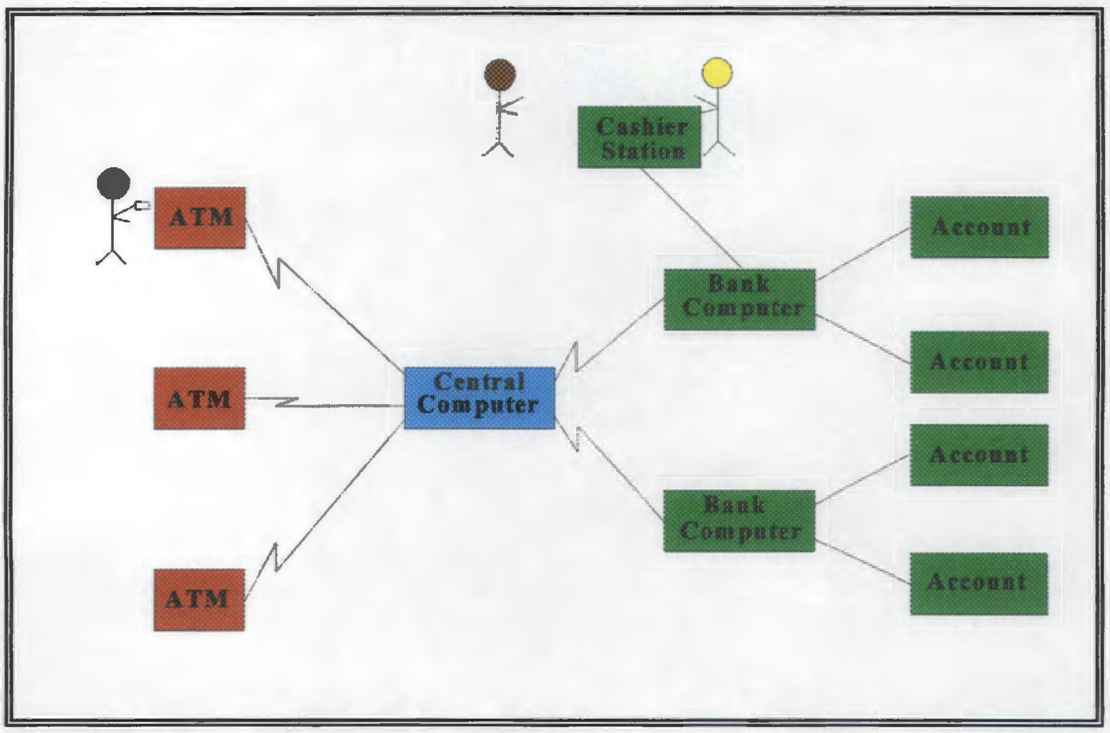


Figure G.1 ATM network (Rumbaugh *et al.*, 1991)

The Analysis Cycle is completed and one of the deliverables, namely the Object Model, is presented in *Figure G.2*. When starting with the Design Cycle, Step 1 is: "Breaking a system into subsystems". This step uses the Object Model.

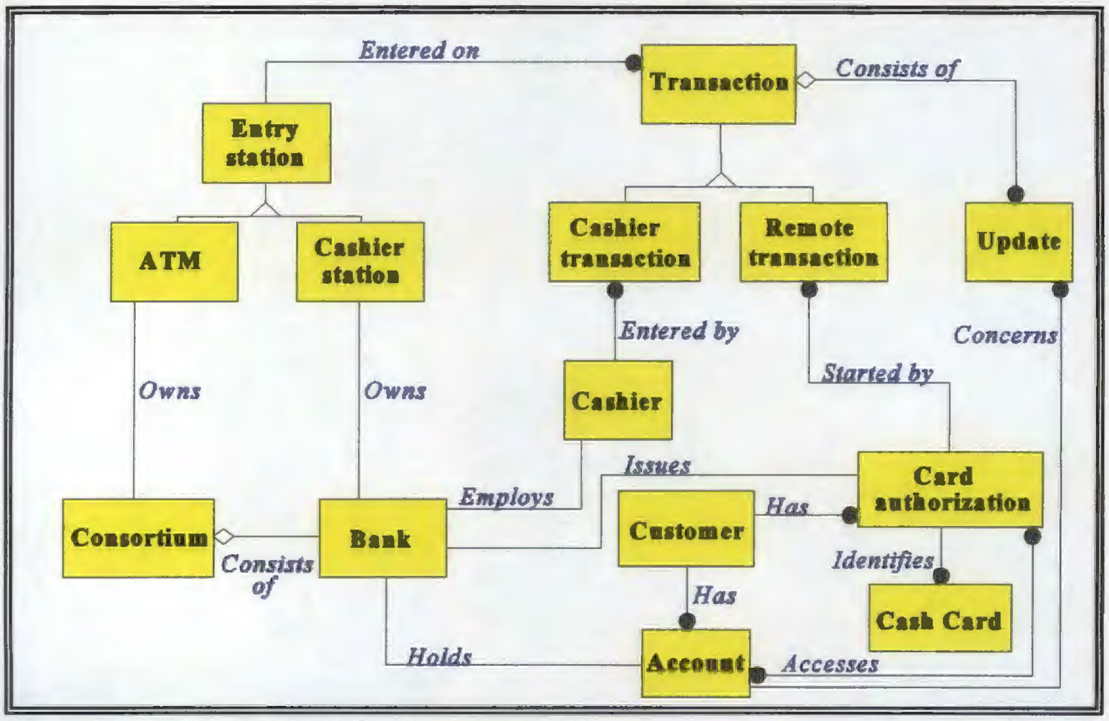


Figure G.2 ATM Object Model (Rumbaugh *et al.*, 1991)

When activating the prototype, the following window appears:

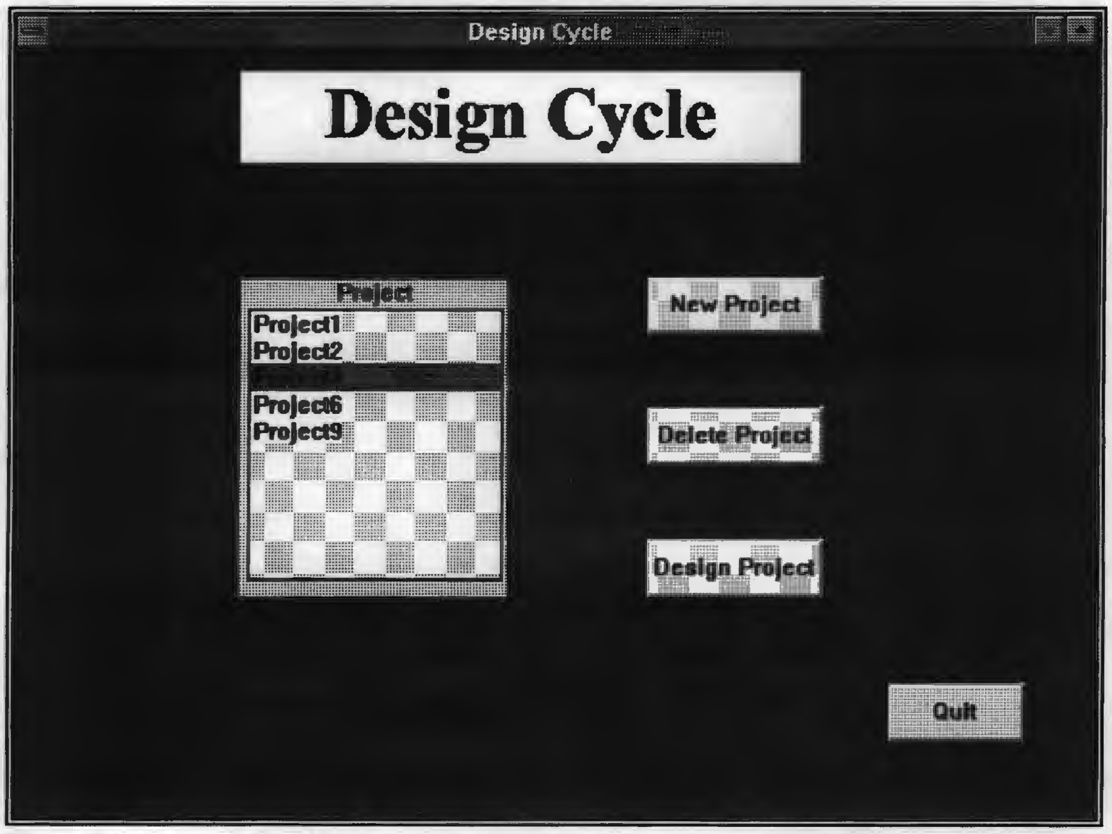
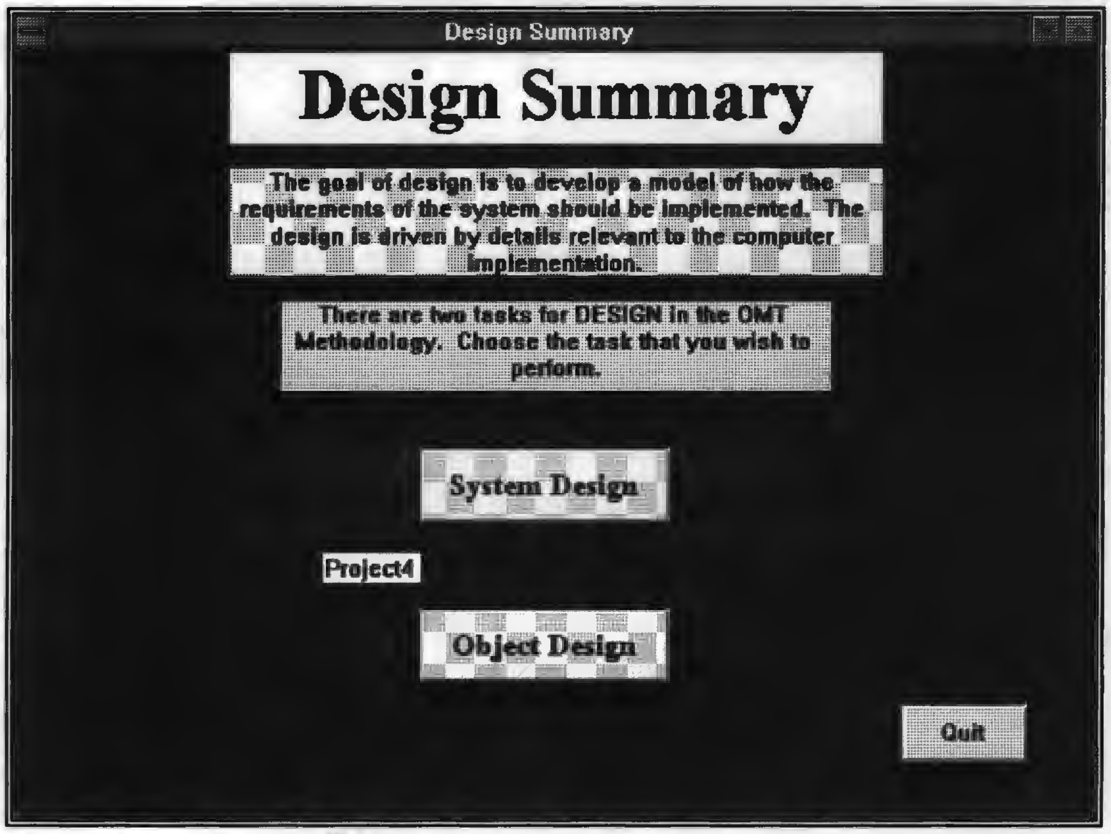


Figure G.3 Design Cycle Window

In the Project box one may select the specific project. To continue with the Design Cycle, click on the Design Project Button.

The next window that appears will be:



Design Summary

Design Summary

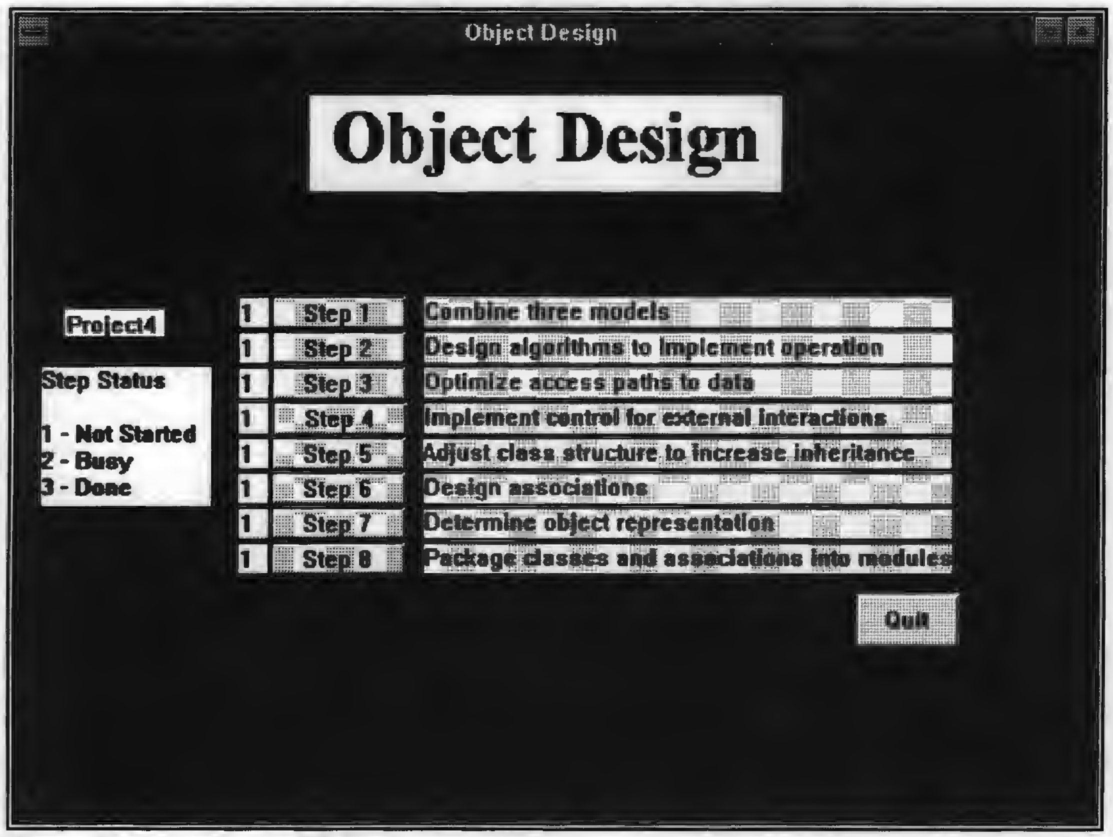
The goal of design is to develop a model of how the requirements of the system should be implemented. The design is driven by details relevant to the computer implementation.

There are two tasks for DESIGN in the OMT Methodology. Choose the task that you wish to perform.

Figure G.4 Design Summary Window

In the OMT methodology there are two main tasks to be completed for Design, namely the System Design task and the Object Design task. Click on System Design for purposes of this demonstration.

If clicked on Object Design, the following window would have appeared:


Object Design

Object Design

Project4

Step Status

1 - Not Started
2 - Busy
3 - Done

1	Step 1	Combine three models
1	Step 2	Design algorithms to implement operation
1	Step 3	Optimize access paths to data
1	Step 4	Implement control for external interactions
1	Step 5	Adjust class structure to increase inheritance
1	Step 6	Design associations
1	Step 7	Determine object representation
1	Step 8	Package classes and associations into modules

Figure G.5 Object Design Window

This window and task is not supported by this prototype.

If clicked on System Design, the following window will appear:

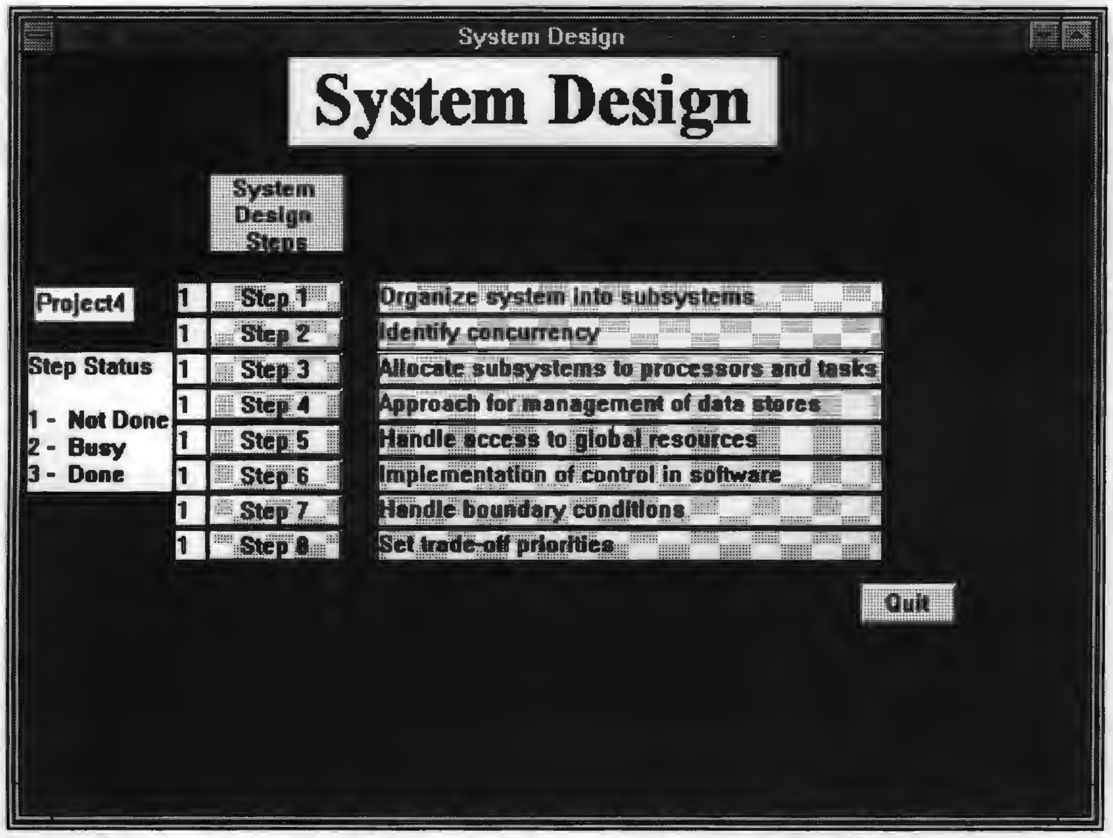

System Design
System Design
System Design Steps
Project4
Step Status
1 - Not Done
2 - Busy
3 - Done
1 Step 1 Organize system into subsystems
1 Step 2 Identify concurrency
1 Step 3 Allocate subsystems to processors and tasks
1 Step 4 Approach for management of data stores
1 Step 5 Handle access to global resources
1 Step 6 Implementation of control in software
1 Step 7 Handle boundary conditions
1 Step 8 Set trade-off priorities
Quit

Figure G.6 System Design Window

If one wants to do one of the eight steps of System Design, then one must click on the appropriate button, for example click on the Step 1 button. Steps 2 to 8 are not supported by this prototype. In the white blocks, next to the Step-buttons, the Step Status for each step can be seen.

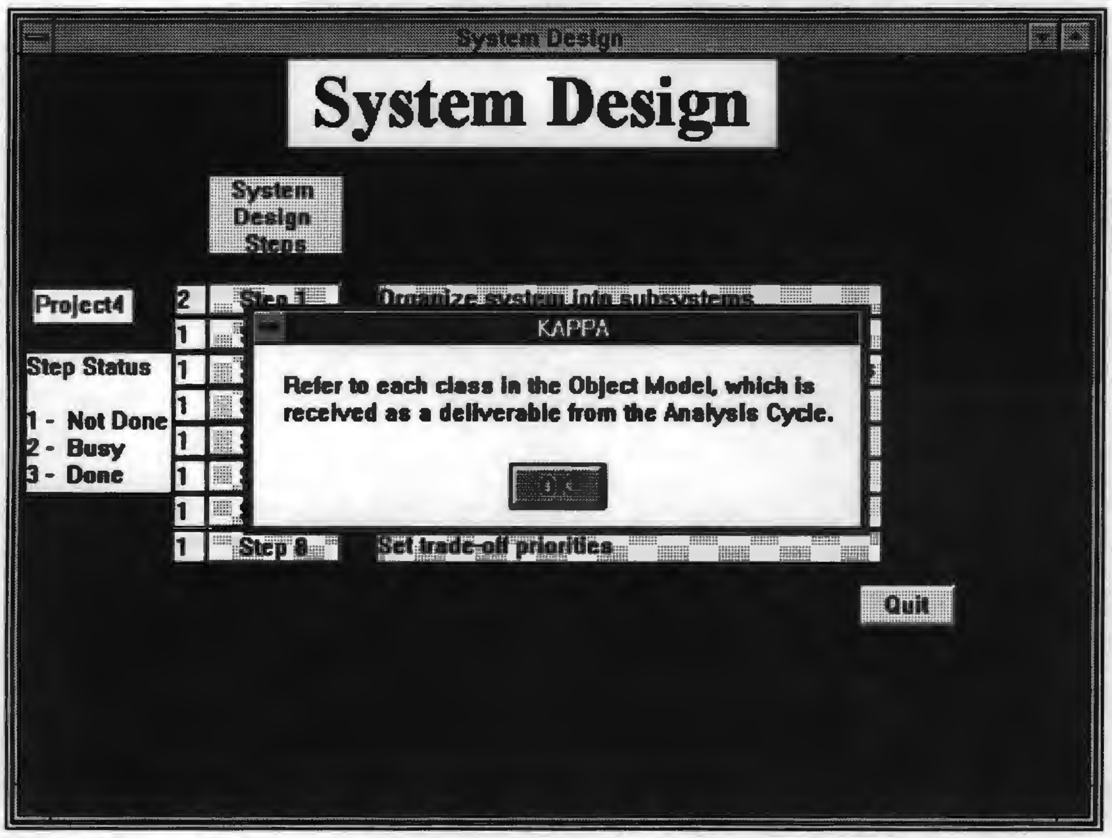
An example of the support which the rules of the knowledge base provide, follows.

When breaking a system into subsystems, each subsystem encompasses facets of the system which share some common grounds. These grounds are firstly execution on the same kind of hardware, secondly hardware in the same physical location and thirdly, similar functionality. This is the reason why the first three questions in the dialogue part of the demonstration confront the novice with the following detail:

Question 1.	*Refer to each class in the Object Model, which is received as a deliverable from the Analysis Cycle. On what type of Hardware component does the class under investigation execute?*
Question 2.	*Indicate the physical location of the hardware component upon which the class under investigation executes.*
Question 3.	*A service is a group of related functions which share some common purpose. Classify the service of the class under investigation.*

According to the answers to these questions, the first iteration for the possible sub-systems for the target system may be completed and the first decisions made. All three questions must be responded to before an inference is derived by the expert system.

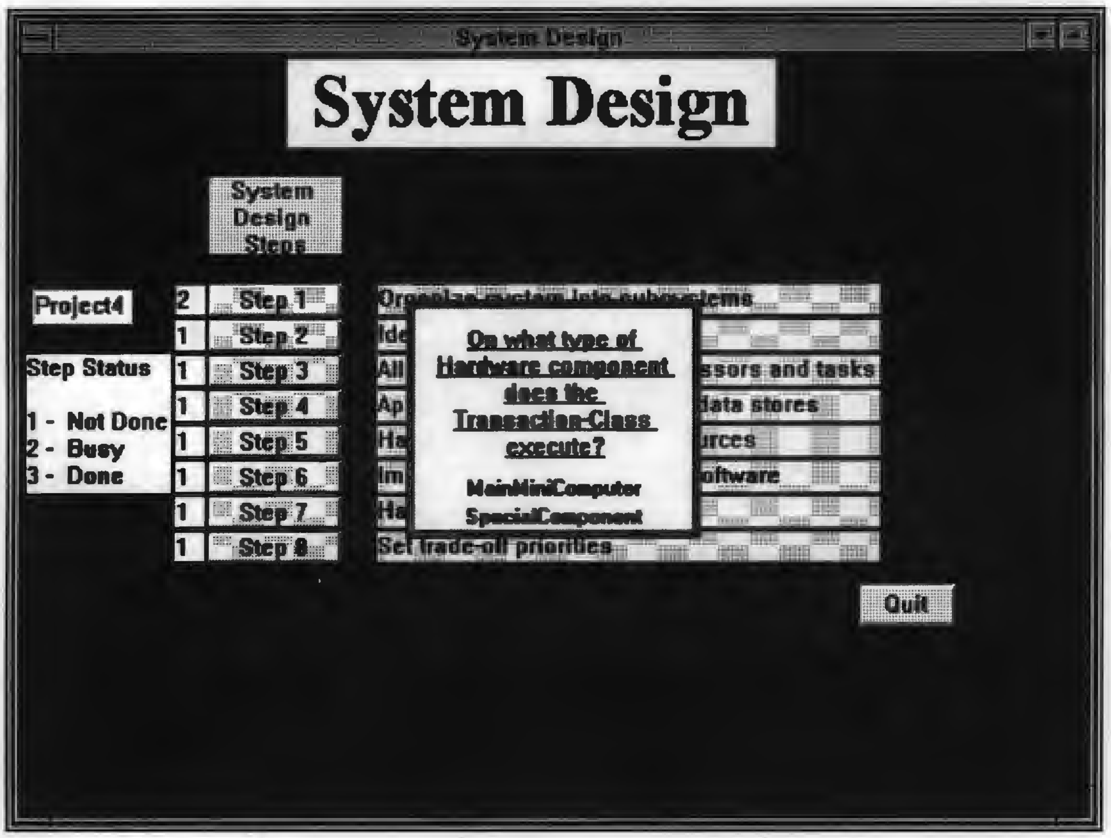
Question 1, Part 1:



Refer to each class in the Object Model, which is received as a deliverable from the Analysis Cycle.

Figure G.7 Question 1.1

Question 1, Part 2:



2	Step 1	Organize system into subsystems
1	Step 2	Ide
1	Step 3	All … ssors and tasks
1	Step 4	Ap … data stores
1	Step 5	Ha … urces
1	Step 6	Im … oftware
1	Step 7	Ha
1	Step 8	Set trade-off priorities

On what type of Hardware component does the Transaction-Class execute?

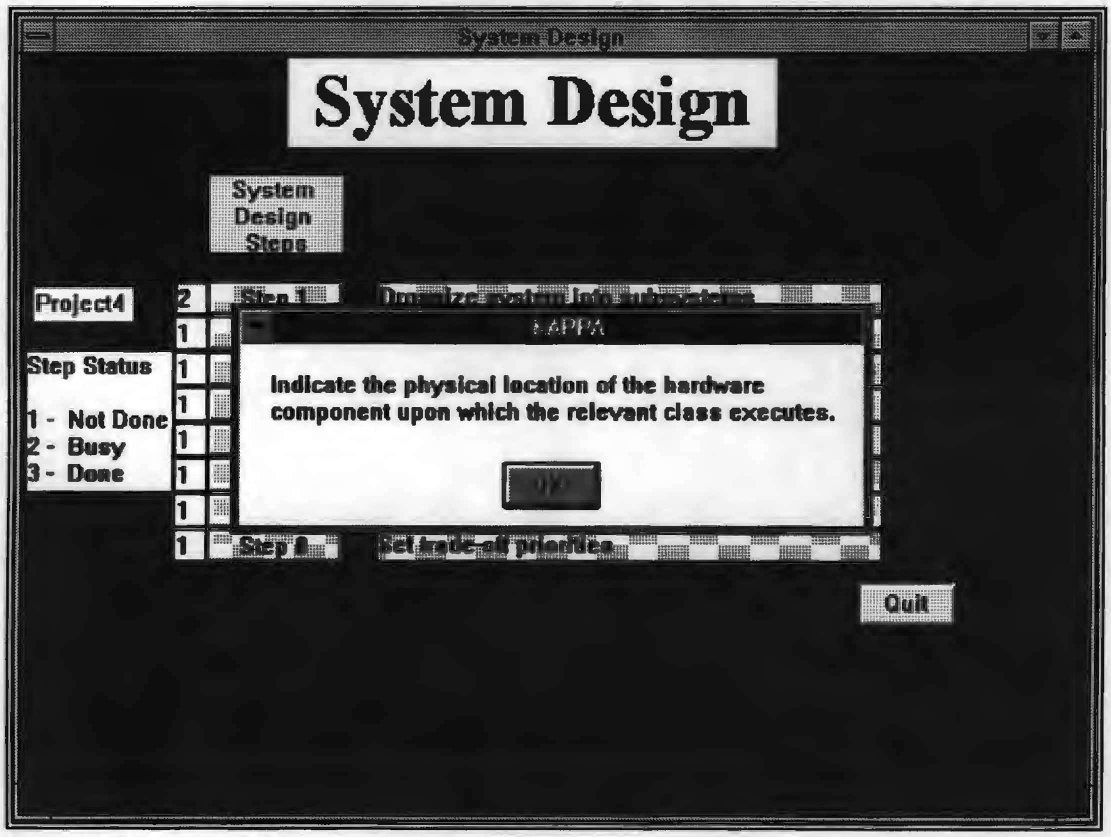
MainMiniComputer

SpecialComponent

Figure G.8 Question 1.2

Question 1, Part 2 is asked for every class in the Object Model.

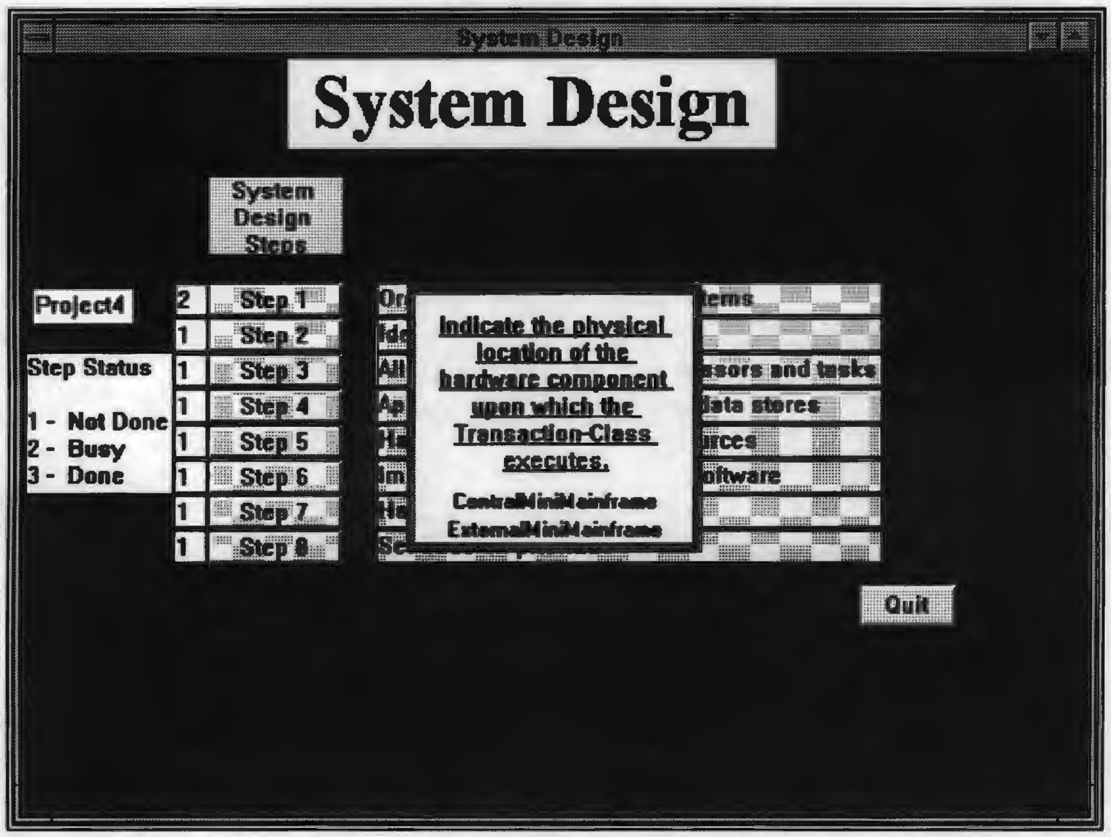
Question 2, Part 1:



Indicate the physical location of the hardware component upon which the relevant class executes.

Figure G.9 Question 2.1

Question 2, Part 2:



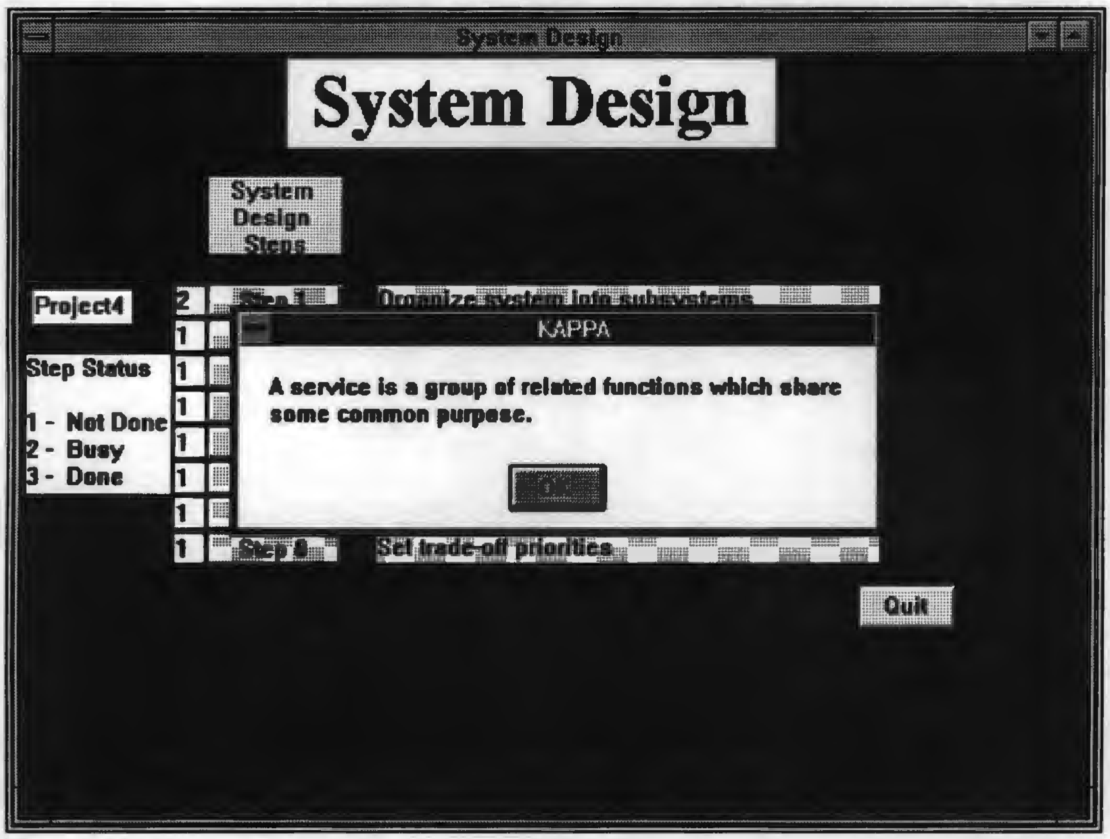
Indicate the physical location of the hardware component upon which the Transaction-Class executes.

CentralMiniMainframe
ExternalMiniMainframe

Figure G.10 Question 2.2

Question 2, Part 2 is asked for every class in the Object Model.

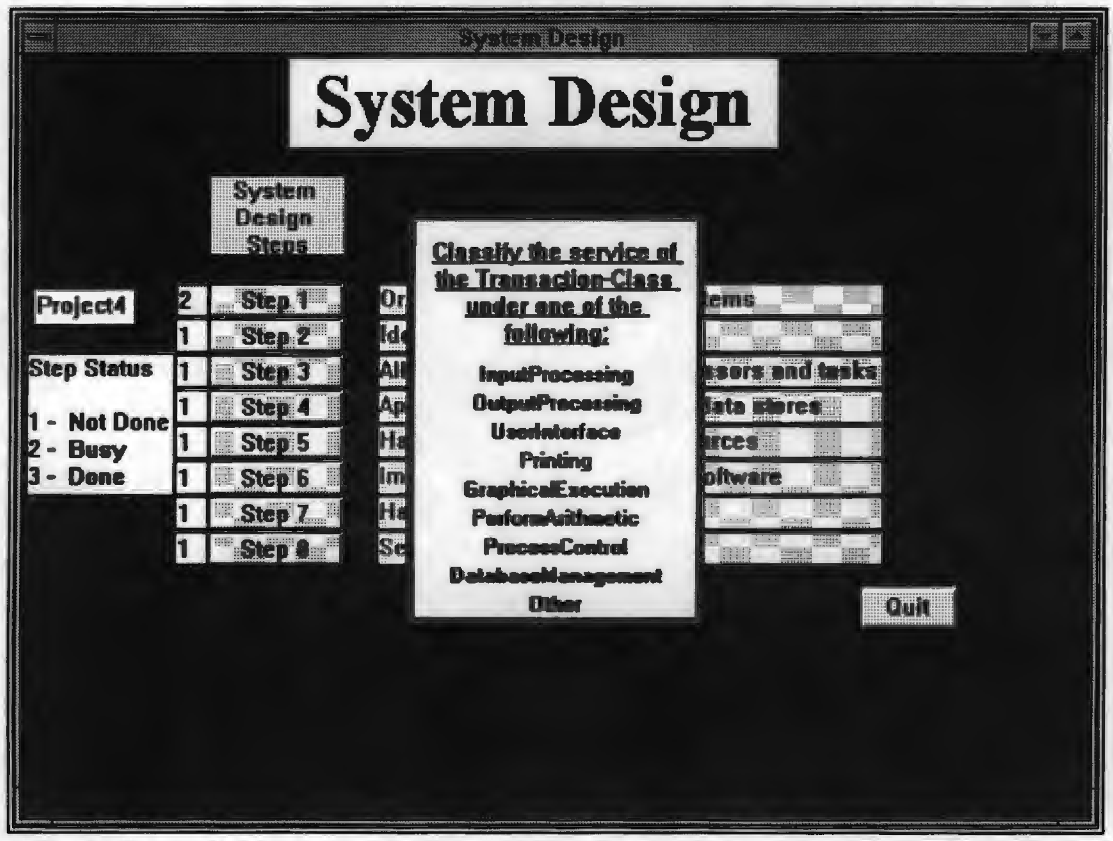
Question 3, Part 1:



A service is a group of related functions which share some common purpose.

Figure G.11 Question 3.1

Question 3, Part 2:



System Design

System Design Steps

Project4

2	Step 1
1	Step 2
1	Step 3
1	Step 4
1	Step 5
1	Step 6
1	Step 7
1	Step 8

Step Status

1 - Not Done
2 - Busy
3 - Done

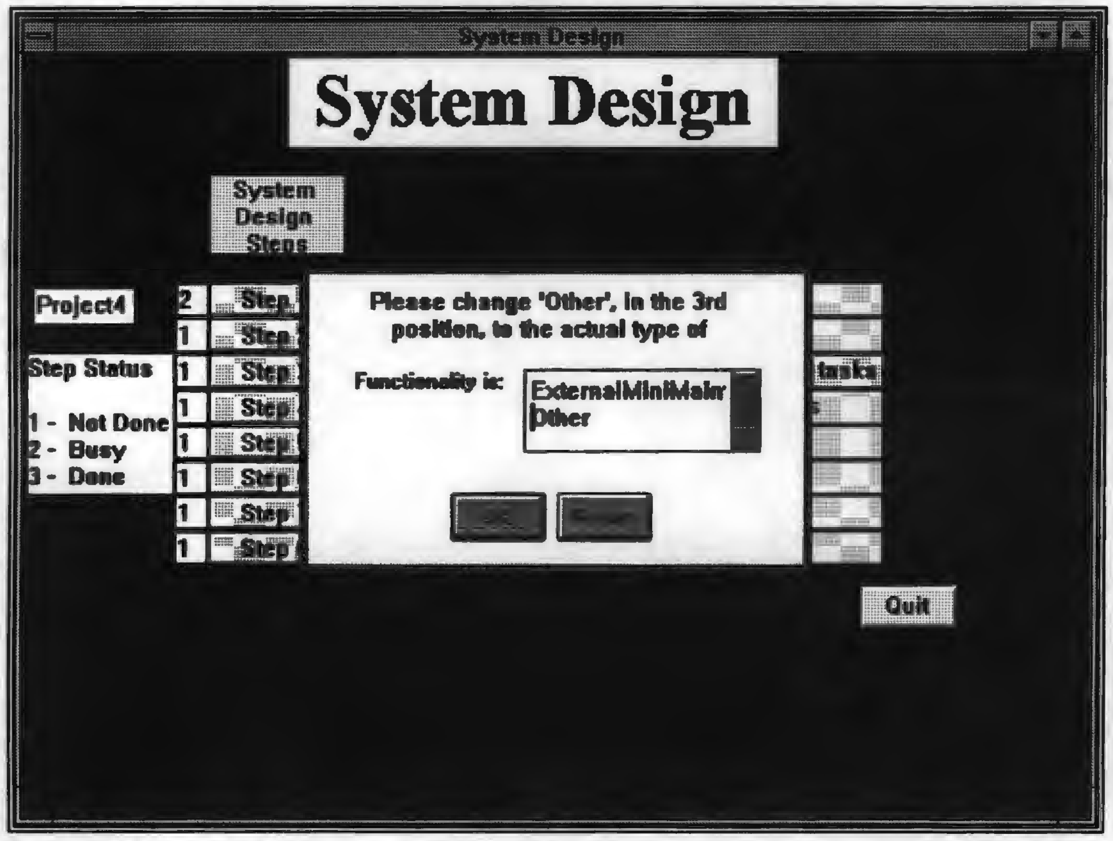
Classify the service of the Transaction-Class under one of the following:

InputProcessing
OutputProcessing
UserInterface
Printing
GraphicalExecution
PerformArithmetic
ProcessControl
DatabaseManagement
Other

Figure G.12 Question 3.2

Question 3, Part 2 is asked for every class in the Object Model. Now the rules are activated and the reasoning process begins.

Question 3, Part 3:



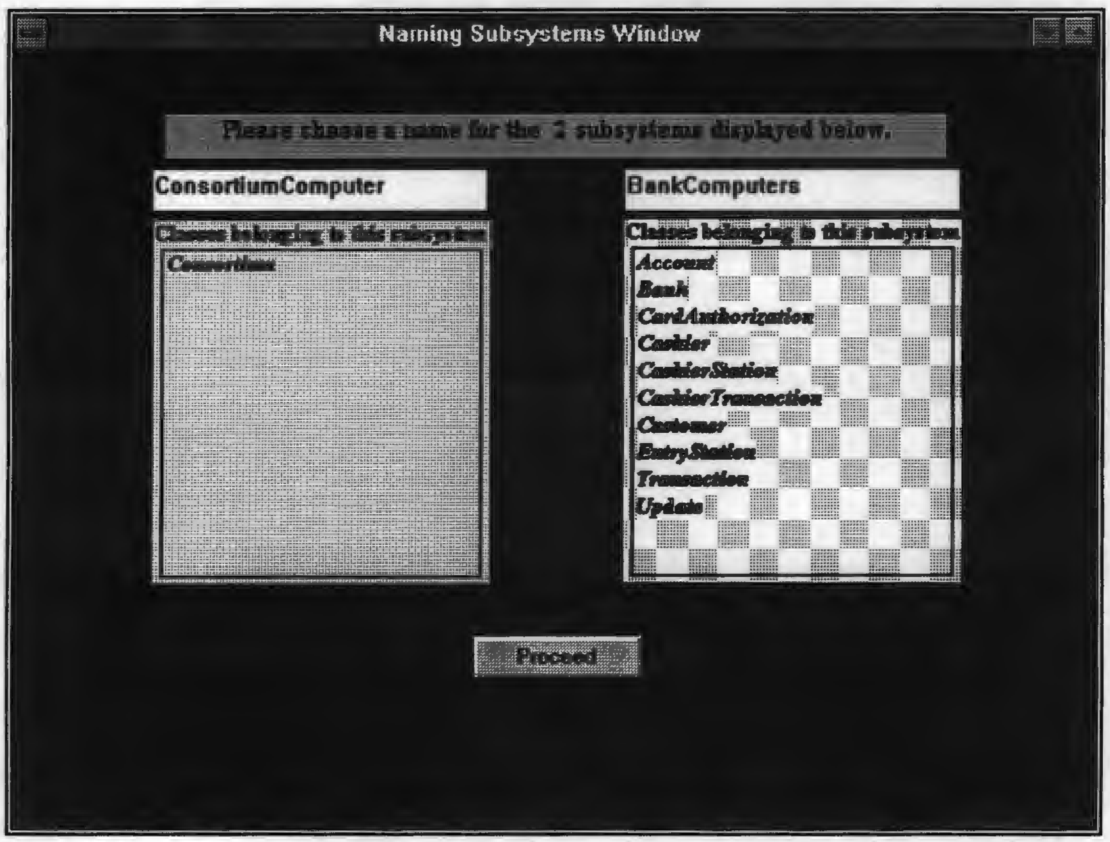
Please change 'Other', in the 3rd position, to the actual type of

Functionality is: ExternalMiniMain
Other

Figure G.13 Question 3.3

For the *RemoteTransaction*-class, *ATM*-class and *CashCard*-class, one must enter ***Other*** when asked to "Classify the service". This window will appear and ***Other*** must be changed to *ATM*.

At this stage, two of the subsystems are without names. They must now be named.



Please choose a name for the 2 subsystems displayed below.

ConsortiumComputer

Classes belonging to this subsystem

Consortium

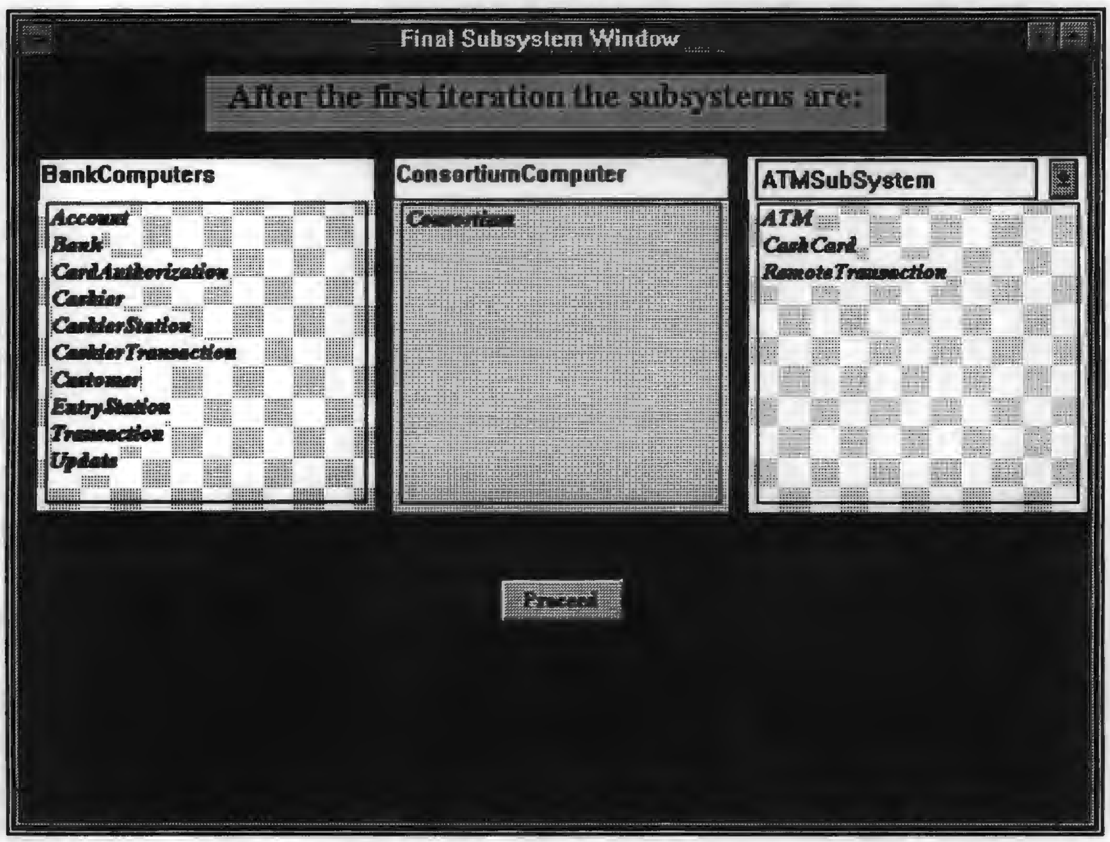
BankComputers

Classes belonging to this subsystem

Account
Bank
CardAuthorization
Cashier
CashierStation
CashierTransaction
Customer
EntryStation
Transaction
Update

Figure G.14 Naming Subsystems Window

This is the first decision which was made after the first iteration of the decision-making process for the possible subsystems.


Final Subsystem Window

After the first iteration the subsystems are:

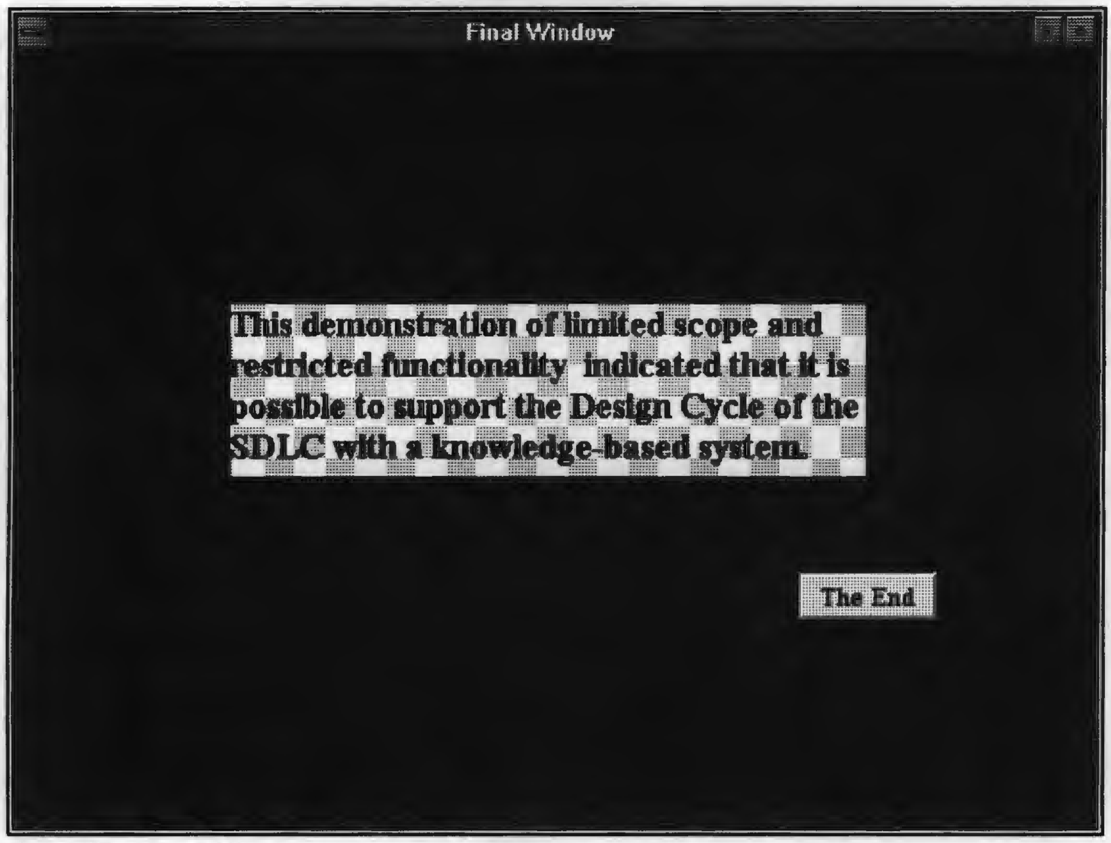
BankComputers
- Account
- Bank
- CardAuthorization
- Cashier
- CashierStation
- CashierTransaction
- Customer
- EntryStation
- Transaction
- Update

ConsortiumComputer
- Consortium

ATMSubSystem
- ATM
- CashCard
- RemoteTransaction

Figure G.15 Final Subsystems Window

The demonstration ends with the following screen.



This demonstration of limited scope and restricted functionality indicated that it is possible to support the Design Cycle of the SDLC with a knowledge-based system.

Figure G.16 Final Window

The source code of this prototype is listed in *Appendix H*.

APPENDIX H

Source Code of the Prototype

The generic source code of classes, subclasses and instances of these, in Kappa-PC, was deliberately omitted in the interests of space, but is available from the author, as well as on the accompanying stiffy in the file: design.kal

```
    /************************************
     **** FUNCTION: NewProjectButtonAction
     ************************************/
MakeFunction( NewProjectButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /************************************
     **** FUNCTION: DeleteProjectButtonAction
     ************************************/
MakeFunction( DeleteProjectButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /************************************
     **** FUNCTION: QuitProjectButtonAction
     ************************************/
MakeFunction( QuitProjectButtonAction, [],
   HideWindow( SESSION ) );

    /************************************
     **** FUNCTION: SystemDesignButtonAction
     ************************************/
MakeFunction( SystemDesignButtonAction, [],
   {
   HideWindow( DesignSummary );
   ShowWindow( SystemDesignMenu );
   } );


    /************************************
     **** FUNCTION: QuitSummaryButtonAction
     ************************************/
MakeFunction( QuitSummaryButtonAction, [],
   {
   HideWindow( DesignSummary );
   ShowWindow( SESSION );
   } );

    /************************************
     **** FUNCTION: DesignProjectButtonAction
     ************************************/
MakeFunction( DesignProjectButtonAction, [],
   If KnownValue?( Design:Project )
     Then {
         EnumList( SystemDesign:ListOfObjects, slot,
            SetValue( ObjectClass, slot, None, None, None ) );
         SetValue( StepSD1:Status, 1 );
         SendMessage( StepSD1, ClearSpecialSlots );
         ResetValue( StepSD1, Test1 );
         ResetValue( StepSD1, Test2 );
         HideWindow( SESSION );
         ShowWindow( DesignSummary );
         }
     Else Beep( ) );

    /************************************
     **** FUNCTION: ObjectDesignButtonAction
     ************************************/
MakeFunction( ObjectDesignButtonAction, [],
   {
   HideWindow( DesignSummary );
   ShowWindow( ObjectDesignMenu );
   } );

    /************************************
     **** FUNCTION: QuitSystemDesignButtonAction
     ************************************/
MakeFunction( QuitSystemDesignButtonAction, [],
   {
   HideWindow( SystemDesignMenu );
   ShowWindow( DesignSummary );
   } );

    /************************************
     **** FUNCTION: QuitObjectDesignButtonAction
     ************************************/
MakeFunction( QuitObjectDesignButtonAction, [],
   {
   HideWindow( ObjectDesignMenu );
   ShowWindow( DesignSummary );
   } );

    /************************************
     **** FUNCTION: SD1ButtonAction
     ************************************/
MakeFunction( SD1ButtonAction, [],
   {
   SetValue( StepSD1:Status, 2 );
   SendMessage( ObjectClass, ListOfObjects );
   SendMessage( ObjectClass, Reset );
   SendMessage( ObjectClass, HardwareQuestion );
   SendMessage( ObjectClass, PhysicalLocationQuestion );
   SendMessage( ObjectClass, FunctionalityQuestion );
   SetValue( StepSD1:Status, 3 );
   SendMessage( StepSD1, PerformDesignStep );
   } );

     /************************************
      **** FUNCTION: DeleteProjectButtonActionBak
      ************************************/
MakeFunction( DeleteProjectButtonActionBak, [],
   If KnownValue?( Design:Project )
     Then {
         SendMessage( Design, DeleteProject );
         ResetImage( ProjectSelection );
         }
     Else Beep( ) );

     /************************************
      **** FUNCTION: NewProjectButtonActionBak
      ************************************/
MakeFunction( NewProjectButtonActionBak, [],
   {
   SendMessage( Design, CreateNewProject );
   ResetImage( ProjectSelection );
   } );

     /************************************
      **** FUNCTION: OD1ButtonAction
      ************************************/
MakeFunction( OD1ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

     /************************************
      **** FUNCTION: OD2ButtonAction
      ************************************/
MakeFunction( OD2ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

     /************************************
      **** FUNCTION: OD3ButtonAction
      ************************************/
MakeFunction( OD3ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );
```

```
    /***********************************
     **** FUNCTION: OD4ButtonAction
     ***********************************/
MakeFunction( OD4ButtonAction, [],
  PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /***********************************
     **** FUNCTION: OD5ButtonAction
     ***********************************/
MakeFunction( OD5ButtonAction, [],
  PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /***********************************
     **** FUNCTION: OD6ButtonAction
     ***********************************/
MakeFunction( OD6ButtonAction, [],
  PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /***********************************
     **** FUNCTION: OD7ButtonAction
     ***********************************/
MakeFunction( OD7ButtonAction, [],
  PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /***********************************
     **** FUNCTION: OD8ButtonAction
     ***********************************/
MakeFunction( OD8ButtonAction, [],
  PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /***********************************
     **** FUNCTION: SD2ButtonAction
     ***********************************/
MakeFunction( SD2ButtonAction, [],
  PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /***********************************
     **** FUNCTION: SD3ButtonAction
     ***********************************/
MakeFunction( SD3ButtonAction, [],
  PostMessage( "This Button is not supported for purposes of this demonstration." ) );

    /***********************************
     **** FUNCTION: SD4ButtonAction
     ***********************************/
MakeFunction( SD4ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

     /*************************************
      **** FUNCTION: SD5ButtonAction
      *************************************/
MakeFunction( SD5ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

     /*************************************
      **** FUNCTION: SD6ButtonAction
      *************************************/
MakeFunction( SD6ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

     /*************************************
      **** FUNCTION: SD7ButtonAction
      *************************************/
MakeFunction( SD7ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

     /*************************************
      **** FUNCTION: SD8ButtonAction
      *************************************/
MakeFunction( SD8ButtonAction, [],
   PostMessage( "This Button is not supported for purposes of this demonstration." ) );

     /*************************************
      **** FUNCTION: ProceedButtonAction
      *************************************/
MakeFunction( ProceedButtonAction, [],
   {
   HideWindow( SubSystemNameWindow );
   SendMessage( SpecialsWindow, ShowWindow );
   } );

     /*************************************
      **** FUNCTION: SpecialsWindowButtonAction
      *************************************/
MakeFunction( SpecialsWindowButtonAction, [],
   {
   HideWindow( SpecialsWindow );
   ShowWindow( FinalWindow );
   } );
```

```
    /************************************
    **** FUNCTION: EndButtonAction
    ************************************/
MakeFunction( EndButtonAction, [],
   {
   HideWindow( FinalWindow );
   HideWindow( SubSystemNameWindow );
   HideWindow( SpecialsWindow );
   HideWindow( ObjectDesignMenu );
   HideWindow( SystemDesignMenu );
   HideWindow( DesignSummary );
   HideWindow( SESSION );
   } );


/************** METHOD: CreateNewProject  **************/
MakeMethod( Design, CreateNewProject, [],
   {
   PostInputForm( "Enter the new project's name:", Global, ProjectName,
      "Project : " );
   Let [name Global:ProjectName]
      {
      If Instance?( name )
        Then PostMessage( FormatValue( "Warning: %s
              already exists.",
                      name ) )
        Else MakeInstance( name, Project );
      Self:Project = name;
      };
   } );

/************** METHOD: DeleteProject  **************/
MakeMethod( Design, DeleteProject, [],
   Let [title FormatValue( "Deleting %s", Self:Project )]
      If ( PostMenu( title, OK, Cancel ) #= OK )
        Then {
           DeleteInstance( Self:Project );
           ResetValue( Self:Project );
           } );

/************** METHOD: QuitProject  **************/
MakeMethod( Design, QuitProject, [],
   HideWindow( DesignCycle ) );

/************** METHOD: ResetSubSystems  **************/
MakeMethod( Design, ResetSubSystems, [],
   {
   SetValue( StepSD1:Status, 1 );
   ResetValue( StepSD1:Test1 );
   ResetValue( StepSD1:Test2 );
   SendMessage( StepSD1, ClearSpecialSlots );
   ResetValue( StepSD1:Specials );
   } );
MakeSlot( Design:Project );
SetSlotOption( Design:Project, VALUE_TYPE, OBJECT );
SetSlotOption( Design:Project, ALLOWABLE_CLASSES, Project );
Design:Project = Project4;
SetSlotOption( Design:Project, IF_NEEDED, NULL );
SetSlotOption( Design:Project, WHEN_ACCESS, NULL );
SetSlotOption( Design:Project, BEFORE_CHANGE, NULL );
SetSlotOption( Design:Project, AFTER_CHANGE, NULL );
SetSlotOption( Design:Project, IMAGE, ProjectSelection, DesignSumEdit, SystemDesignEdit,
ObjectDesignEdit );


/************** METHOD: PerformDesignStep **************/
MakeMethod( SystemDesign, PerformDesignStep, [],
   {
   EnumList( SystemDesign:ListOfObjects, object,
      {
      Global:Object = object;
      Assert( Global:Object );
      SetForwardChainMode( BREADTHFIRST );
      ForwardChain( NULL, Self:SDRuleSet );
      } );
   ShowWindow( SubSystemNameWindow );
   } );
MakeSlot( SystemDesign:SystemDesignSteps );
SetSlotOption( SystemDesign:SystemDesignSteps, ALLOWABLE_VALUES, Step1, Step2, Step3,
Step4, Step5, Step6, Step7, Step8 );
SystemDesign:SystemDesignSteps = Step5;
SetSlotOption( SystemDesign:SystemDesignSteps, IF_NEEDED, NULL );
SetSlotOption( SystemDesign:SystemDesignSteps, WHEN_ACCESS, NULL );
SetSlotOption( SystemDesign:SystemDesignSteps, BEFORE_CHANGE, NULL );
SetSlotOption( SystemDesign:SystemDesignSteps, AFTER_CHANGE, NULL );
MakeSlot( SystemDesign:ListOfObjects );
SetSlotOption( SystemDesign:ListOfObjects, MULTIPLE );
SetValue( SystemDesign:ListOfObjects, Transaction, EntryStation, CashierTransaction,
RemoteTransaction, Update, ATM, CashierStation, Cashier, CardAuthorization, Customer,
Consortium, Bank, Account, CashCard );
SetSlotOption( SystemDesign:ListOfObjects, IF_NEEDED, NULL );
SetSlotOption( SystemDesign:ListOfObjects, WHEN_ACCESS, NULL );
SetSlotOption( SystemDesign:ListOfObjects, BEFORE_CHANGE, NULL );
SetSlotOption( SystemDesign:ListOfObjects, AFTER_CHANGE, NULL );

/************** METHOD: Reset  **************/
MakeMethod( ObjectClass, Reset, [],
  {
  QuestionsWork:CurrentObjectClassNumber = 1;
  EnumList( SystemDesign:ListOfObjects, slot,
    {
    If GetSlotOption( ObjectClass:slot, MULTIPLE )
      Then SetValue( ObjectClass, slot, None, None, None );
    QuestionsWork:CurrentObjectClassNumber += 1;
    } );
  } );

/************** METHOD: HardwareQuestion  **************/
MakeMethod( ObjectClass, HardwareQuestion, [],
  {
  PostMessage( "Refer to each class in the Object Model, which is received as a deliverable from
the Analysis Cycle." );
  QuestionsWork:CurrentObjectClassNumber = 1;
  EnumList( SystemDesign:ListOfObjects, slot,
    {
    If GetSlotOption( ObjectClass:slot, MULTIPLE )
      Then SetNthElem( ObjectClass:slot, 1, PostMenu( "On what type of Hardware component
does the "
                                                  #
                                                  GetNthElem( SystemDesign:ListOfObjects,
                                                    QuestionsWork:CurrentObjectClassNumber )
                                                  #
                                                  "-Class execute?",
                                          MainMiniComputer,
                                          SpecialComponent ) );
    QuestionsWork:CurrentObjectClassNumber += 1;
    If ( GetNthElem( ObjectClass:slot, 1 ) #= SSpecialComponent )
      Then PostInputForm( "Please change 'SpecialComponent' to the actual type of Hardware
Component:",
              ObjectClass:slot, "Harware Component is:" );
    } );
  } );

/************** METHOD: PhysicalLocationQuestion  **************/
MakeMethod( ObjectClass, PhysicalLocationQuestion, [],
  {
  PostMessage( "Indicate the physical location of the hardware component which the relevant class
executes on." );
```

```
   QuestionsWork:CurrentObjectClassNumber = 1;
   EnumList( SystemDesign:ListOfObjects, slot,
     {
     If GetSlotOption( ObjectClass:slot, MULTIPLE )
       Then SetNthElem( ObjectClass:slot, 2, PostMenu( "Indicate the physical location of the
hardware component which the "
                                          #
                                          GetNthElem( SystemDesign:ListOfObjects,
                                             QuestionsWork:CurrentObjectClassNumber )
                                          #
                                          "-Class executes on.",
                                   CentralMiniMainframe,
                                   ExternalMiniMainframe ) );
     QuestionsWork:CurrentObjectClassNumber += 1;
     If ( GetNthElem( ObjectClass:slot, 2 ) #= SpecialComponent )
       Then PostInputForm( "Please change 'SpecialComponent' in the 2nd position, to the correct
physical location:",
             ObjectClass:slot, "Physical Location is:" );
     } );
   } );

/************** METHOD: FunctionalityQuestion  **************/
MakeMethod( ObjectClass, FunctionalityQuestion, [],
   {
   PostMessage( "A service is a group of related functions which share some common purpose." );
   QuestionsWork:CurrentObjectClassNumber = 1;
   EnumList( SystemDesign:ListOfObjects, slot,
     {
     If GetSlotOption( ObjectClass:slot, MULTIPLE )
       Then SetNthElem( ObjectClass:slot, 3, PostMenu( "Classify the service of the "
                                          #
                                          GetNthElem( SystemDesign:ListOfObjects,
                                             QuestionsWork:CurrentObjectClassNumber )
                                          #
                                          "-Class under one of the following:",
                                   InputProcessing,
                                   OutputProcessing,
                                   UserInterface,
                                   Printing, GraphicalExecution,
                                   PerformArithmetic,
                                   ProcessControl,
                                   DatabaseManagement,
                                   Other ) );
     QuestionsWork:CurrentObjectClassNumber += 1;
     If ( GetNthElem( ObjectClass:slot, 3 ) #= Other )
       Then PostInputForm( "Please change 'Other', in the 3rd position, to the actual type of
Functionality:",
             ObjectClass:slot, "Functionality is: " );
     } );
  } );

/************* METHOD: ListOfObjects  *************/
MakeMethod( ObjectClass, ListOfObjects, [],
  {
  If Slot?( SystemDesign:ListOfObjects )
    Then {
        ResetValue( SystemDesign:ListOfObjects );
        SetSlotOption( SystemDesign:ListOfObjects, MULTIPLE );
        }
    Else {
        MakeSlot( SystemDesign:ListOfObjects );
        SetSlotOption( SystemDesign:ListOfObjects, MULTIPLE );
        };
  GetSlotList( ObjectClass, SystemDesign:ListOfObjects );
  } );
MakeSlot( ObjectClass:Transaction );
SetSlotOption( ObjectClass:Transaction, MULTIPLE );
SetValue( ObjectClass:Transaction, MainMiniComputer, CentralMiniMainframe, PerformArithmetic
);
SetSlotOption( ObjectClass:Transaction, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:Transaction, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:Transaction, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:Transaction, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:EntryStation );
SetSlotOption( ObjectClass:EntryStation, MULTIPLE );
SetValue( ObjectClass:EntryStation, MainMiniComputer, CentralMiniMainframe, UserInterface );
SetSlotOption( ObjectClass:EntryStation, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:EntryStation, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:EntryStation, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:EntryStation, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:CashierTransaction );
SetSlotOption( ObjectClass:CashierTransaction, MULTIPLE );
SetValue( ObjectClass:CashierTransaction, MainMiniComputer, CentralMiniMainframe,
InputProcessing );
SetSlotOption( ObjectClass:CashierTransaction, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:CashierTransaction, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:CashierTransaction, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:CashierTransaction, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:RemoteTransaction );
SetSlotOption( ObjectClass:RemoteTransaction, MULTIPLE );
SetValue( ObjectClass:RemoteTransaction, SpecialComponent, ExternalMiniMainframe, ATM );
SetSlotOption( ObjectClass:RemoteTransaction, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:RemoteTransaction, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:RemoteTransaction, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:RemoteTransaction, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:Update );
SetSlotOption( ObjectClass:Update, MULTIPLE );
SetValue( ObjectClass:Update, MainMiniComputer, CentralMiniMainframe, PerformArithmetic );
SetSlotOption( ObjectClass:Update, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:Update, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:Update, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:Update, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:ATM );
SetSlotOption( ObjectClass:ATM, MULTIPLE );
SetValue( ObjectClass:ATM, SpecialComponent, ExternalMiniMainframe, ATM );
SetSlotOption( ObjectClass:ATM, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:ATM, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:ATM, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:ATM, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:CashierStation );
SetSlotOption( ObjectClass:CashierStation, MULTIPLE );
SetValue( ObjectClass:CashierStation, MainMiniComputer, CentralMiniMainframe, UserInterface );
SetSlotOption( ObjectClass:CashierStation, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:CashierStation, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:CashierStation, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:CashierStation, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:Cashier );
SetSlotOption( ObjectClass:Cashier, MULTIPLE );
SetValue( ObjectClass:Cashier, MainMiniComputer, CentralMiniMainframe, UserInterface );
SetSlotOption( ObjectClass:Cashier, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:Cashier, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:Cashier, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:Cashier, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:CardAuthorization );
SetSlotOption( ObjectClass:CardAuthorization, MULTIPLE );
SetValue(   ObjectClass:CardAuthorization,   MainMiniComputer,   CentralMiniMainframe,
PerformArithmetic );
SetSlotOption( ObjectClass:CardAuthorization, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:CardAuthorization, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:CardAuthorization, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:CardAuthorization, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:Customer );
SetSlotOption( ObjectClass:Customer, MULTIPLE );
SetValue( ObjectClass:Customer, MainMiniComputer, CentralMiniMainframe, UserInterface );
SetSlotOption( ObjectClass:Customer, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:Customer, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:Customer, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:Customer, AFTER_CHANGE, NULL );
```

```
MakeSlot( ObjectClass:Consortium );
SetSlotOption( ObjectClass:Consortium, MULTIPLE );
SetValue(ObjectClass:Consortium,MainMiniComputer,ExternalMiniMainframe,PerformArithmetic
);
SetSlotOption( ObjectClass:Consortium, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:Consortium, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:Consortium, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:Consortium, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:Bank );
SetSlotOption( ObjectClass:Bank, MULTIPLE );
SetValue( ObjectClass:Bank, MainMiniComputer, CentralMiniMainframe, PerformArithmetic );
SetSlotOption( ObjectClass:Bank, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:Bank, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:Bank, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:Bank, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:Account );
SetSlotOption( ObjectClass:Account, MULTIPLE );
SetValue( ObjectClass:Account, MainMiniComputer, CentralMiniMainframe, PerformArithmetic );
SetSlotOption( ObjectClass:Account, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:Account, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:Account, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:Account, AFTER_CHANGE, NULL );
MakeSlot( ObjectClass:CashCard );
SetSlotOption( ObjectClass:CashCard, MULTIPLE );
SetValue( ObjectClass:CashCard, SpecialComponent, ExternalMiniMainframe, ATM );
SetSlotOption( ObjectClass:CashCard, IF_NEEDED, NULL );
SetSlotOption( ObjectClass:CashCard, WHEN_ACCESS, NULL );
SetSlotOption( ObjectClass:CashCard, BEFORE_CHANGE, NULL );
SetSlotOption( ObjectClass:CashCard, AFTER_CHANGE, NULL );


/************** METHOD: ClearSpecialSlots   **************/
MakeMethod( StepSD1, ClearSpecialSlots, [],
   {
   EnumList( Self:Specials, slot, DeleteSlot( Self:slot ) );
   ResetValue( StepSD1:Specials );
   } );
MakeSlot( StepSD1:Status );
SetSlotOption( StepSD1:Status, INHERIT, FALSE );
SetSlotOption( StepSD1:Status, VALUE_TYPE, NUMBER );
StepSD1:Status = 3;
SetSlotOption( StepSD1:Status, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Status, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Status, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Status, AFTER_CHANGE, NULL );
SetSlotOption( StepSD1:Status, IMAGE, SDStep1Edit );
```

```
MakeSlot( StepSD1:SDRuleSet );
SetSlotOption( StepSD1:SDRuleSet, INHERIT, FALSE );
SetSlotOption( StepSD1:SDRuleSet, MULTIPLE );
SetValue( StepSD1:SDRuleSet, RuleTest1, RuleTest2, RuleTest3, Rule4, Rule5, Rule6 );
SetSlotOption( StepSD1:SDRuleSet, IF_NEEDED, NULL );
SetSlotOption( StepSD1:SDRuleSet, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:SDRuleSet, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:SDRuleSet, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Account );
SetSlotOption( StepSD1:Account, INHERIT, FALSE );
SetSlotOption( StepSD1:Account, MULTIPLE );
SetValue( StepSD1:Account, Test1, None, None );
SetSlotOption( StepSD1:Account, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Account, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Account, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Account, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Transaction );
SetSlotOption( StepSD1:Transaction, INHERIT, FALSE );
SetSlotOption( StepSD1:Transaction, MULTIPLE );
SetValue( StepSD1:Transaction, Test1, None, None );
SetSlotOption( StepSD1:Transaction, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Transaction, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Transaction, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Transaction, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:EntryStation );
SetSlotOption( StepSD1:EntryStation, INHERIT, FALSE );
SetSlotOption( StepSD1:EntryStation, MULTIPLE );
SetValue( StepSD1:EntryStation, Test1, None, None );
SetSlotOption( StepSD1:EntryStation, IF_NEEDED, NULL );
SetSlotOption( StepSD1:EntryStation, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:EntryStation, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:EntryStation, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:CashierTransaction );
SetSlotOption( StepSD1:CashierTransaction, INHERIT, FALSE );
SetSlotOption( StepSD1:CashierTransaction, MULTIPLE );
SetValue( StepSD1:CashierTransaction, Test1, None, None );
SetSlotOption( StepSD1:CashierTransaction, IF_NEEDED, NULL );
SetSlotOption( StepSD1:CashierTransaction, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:CashierTransaction, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:CashierTransaction, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:RemoteTransaction );
SetSlotOption( StepSD1:RemoteTransaction, INHERIT, FALSE );
SetSlotOption( StepSD1:RemoteTransaction, MULTIPLE );
SetValue( StepSD1:RemoteTransaction, None, None, ATM );
SetSlotOption( StepSD1:RemoteTransaction, IF_NEEDED, NULL );
SetSlotOption( StepSD1:RemoteTransaction, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:RemoteTransaction, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:RemoteTransaction, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Update );
SetSlotOption( StepSD1:Update, INHERIT, FALSE );
SetSlotOption( StepSD1:Update, MULTIPLE );
SetValue( StepSD1:Update, Test1, None, None );
SetSlotOption( StepSD1:Update, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Update, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Update, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Update, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:ATM );
SetSlotOption( StepSD1:ATM, INHERIT, FALSE );
SetSlotOption( StepSD1:ATM, MULTIPLE );
SetValue( StepSD1:ATM, None, None, ATM );
SetSlotOption( StepSD1:ATM, IF_NEEDED, NULL );
SetSlotOption( StepSD1:ATM, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:ATM, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:ATM, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:CashierStation );
SetSlotOption( StepSD1:CashierStation, INHERIT, FALSE );
SetSlotOption( StepSD1:CashierStation, MULTIPLE );
SetValue( StepSD1:CashierStation, Test1, None, None );
SetSlotOption( StepSD1:CashierStation, IF_NEEDED, NULL );
SetSlotOption( StepSD1:CashierStation, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:CashierStation, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:CashierStation, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Cashier );
SetSlotOption( StepSD1:Cashier, INHERIT, FALSE );
SetSlotOption( StepSD1:Cashier, MULTIPLE );
SetValue( StepSD1:Cashier, Test1, None, None );
SetSlotOption( StepSD1:Cashier, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Cashier, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Cashier, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Cashier, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Customer );
SetSlotOption( StepSD1:Customer, INHERIT, FALSE );
SetSlotOption( StepSD1:Customer, MULTIPLE );
SetValue( StepSD1:Customer, Test1, None, None );
SetSlotOption( StepSD1:Customer, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Customer, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Customer, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Customer, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:CardAuthorization );
SetSlotOption( StepSD1:CardAuthorization, INHERIT, FALSE );
SetSlotOption( StepSD1:CardAuthorization, MULTIPLE );
SetValue( StepSD1:CardAuthorization, Test1, None, None );
```

```
SetSlotOption( StepSD1:CardAuthorization, IF_NEEDED, NULL );
SetSlotOption( StepSD1:CardAuthorization, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:CardAuthorization, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:CardAuthorization, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Consortium );
SetSlotOption( StepSD1:Consortium, INHERIT, FALSE );
SetSlotOption( StepSD1:Consortium, MULTIPLE );
SetValue( StepSD1:Consortium, None, Test2, None );
SetSlotOption( StepSD1:Consortium, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Consortium, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Consortium, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Consortium, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Bank );
SetSlotOption( StepSD1:Bank, INHERIT, FALSE );
SetSlotOption( StepSD1:Bank, MULTIPLE );
SetValue( StepSD1:Bank, Test1, None, None );
SetSlotOption( StepSD1:Bank, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Bank, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Bank, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Bank, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:CashCard );
SetSlotOption( StepSD1:CashCard, INHERIT, FALSE );
SetSlotOption( StepSD1:CashCard, MULTIPLE );
SetValue( StepSD1:CashCard, None, None, ATM );
SetSlotOption( StepSD1:CashCard, IF_NEEDED, NULL );
SetSlotOption( StepSD1:CashCard, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:CashCard, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:CashCard, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:Test1 );
SetSlotOption( StepSD1:Test1, INHERIT, FALSE );
SetSlotOption( StepSD1:Test1, MULTIPLE );
SetValue( StepSD1:Test1, Transaction, EntryStation, CashierTransaction, Update, CashierStation,
Cashier, CardAuthorization, Customer, Bank, Account );
SetSlotOption( StepSD1:Test1, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Test1, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Test1, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Test1, AFTER_CHANGE, NULL );
SetSlotOption( StepSD1:Test1, IMAGE, Test1MultipleListBox, InternalSubSystemNameMultiple );
MakeSlot( StepSD1:Test2 );
SetSlotOption( StepSD1:Test2, INHERIT, FALSE );
SetSlotOption( StepSD1:Test2, MULTIPLE );
SetValue( StepSD1:Test2, Consortium );
SetSlotOption( StepSD1:Test2, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Test2, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Test2, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Test2, AFTER_CHANGE, NULL );
SetSlotOption( StepSD1:Test2, IMAGE, Test2MultipleListBox, ExternalSubSystemNameMultiple );
MakeSlot( StepSD1:Specials );
SetSlotOption( StepSD1:Specials, INHERIT, FALSE );
SetSlotOption( StepSD1:Specials, MULTIPLE );
SetValue( StepSD1:Specials, ATMSubSystem );
SetSlotOption( StepSD1:Specials, IF_NEEDED, NULL );
SetSlotOption( StepSD1:Specials, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:Specials, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:Specials, AFTER_CHANGE, NULL );
MakeSlot( StepSD1:InternalSubSystemName );
SetSlotOption( StepSD1:InternalSubSystemName, INHERIT, FALSE );
StepSD1:InternalSubSystemName = BankComputers;
SetSlotOption( StepSD1:InternalSubSystemName, IF_NEEDED, NULL );
SetSlotOption( StepSD1:InternalSubSystemName, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:InternalSubSystemName, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:InternalSubSystemName, AFTER_CHANGE, NULL );
SetSlotOption(   StepSD1:InternalSubSystemName,    IMAGE,    InternalSubSystemNameEdit,
InternalEditBox );
MakeSlot( StepSD1:ExternalSubSystemName );
SetSlotOption( StepSD1:ExternalSubSystemName, INHERIT, FALSE );
StepSD1:ExternalSubSystemName = ConsortiumComputer;
SetSlotOption( StepSD1:ExternalSubSystemName, IF_NEEDED, NULL );
SetSlotOption( StepSD1:ExternalSubSystemName, WHEN_ACCESS, NULL );
SetSlotOption( StepSD1:ExternalSubSystemName, BEFORE_CHANGE, NULL );
SetSlotOption( StepSD1:ExternalSubSystemName, AFTER_CHANGE, NULL );
SetSlotOption(   StepSD1:ExternalSubSystemName,    IMAGE,    ExternalSubSystemNameEdit,
ExternalEditBox );
StepSD1:SystemDesignSteps = Step1;
MakeSlot( StepSD1:ATMSubSystem );
SetSlotOption( StepSD1:ATMSubSystem, MULTIPLE );
SetValue( StepSD1:ATMSubSystem, RemoteTransaction, ATM, CashCard );


/************** METHOD: ShowWindow  **************/
MakeMethod( SpecialsWindow, ShowWindow, [],
   {
   SetValue( Special1ComboBox:AllowableValues, StepSD1:Specials );
   ResetImage( Special1ComboBox );
   ResetImage( Test1MultipleListBox );
   ResetImage( Test2MultipleListBox );
   ShowWindow( Self );
   } );

/************** METHOD: AfterSpecialSelected  **************/
MakeMethod( SpecialsWindow, AfterSpecialSelected, [slotname oldvalue ],
   Let [listbox slotname # ObjectListBox]
```

```
        {
        SetValue( listbox:AllowableValues, GetValue( StepSD1, Self:slotname ) );
        ResetImage( listbox );
        } );
SpecialsWindow:X = 0;
SpecialsWindow:Y = 0;
SpecialsWindow:Title = "Final Subsystem Window";
SpecialsWindow:SessionNumber = 4;
SpecialsWindow:Width = 640;
SpecialsWindow:Height = 480;
SpecialsWindow:Visible = FALSE;
SpecialsWindow:State = HIDDEN;
MakeSlot( SpecialsWindow:Special1 );
SetSlotOption( SpecialsWindow:Special1, INHERIT, FALSE );
SpecialsWindow:Special1 = ATMSubSystem;
SetSlotOption( SpecialsWindow:Special1, IF_NEEDED, NULL );
SetSlotOption( SpecialsWindow:Special1, WHEN_ACCESS, NULL );
SetSlotOption( SpecialsWindow:Special1, BEFORE_CHANGE, NULL );
SetSlotOption( SpecialsWindow:Special1, AFTER_CHANGE, AfterSpecialSelected );
SetSlotOption( SpecialsWindow:Special1, IMAGE, Special1ComboBox );
MakeSlot( SpecialsWindow:Special2 );
SetSlotOption( SpecialsWindow:Special2, INHERIT, FALSE );
SpecialsWindow:Special2 = UserInterfaceSubSystem;
SetSlotOption( SpecialsWindow:Special2, IF_NEEDED, NULL );
SetSlotOption( SpecialsWindow:Special2, WHEN_ACCESS, NULL );
SetSlotOption( SpecialsWindow:Special2, BEFORE_CHANGE, NULL );
SetSlotOption( SpecialsWindow:Special2, AFTER_CHANGE, AfterSpecialSelected );
MakeSlot( SpecialsWindow:Special1Object );
SetSlotOption( SpecialsWindow:Special1Object, INHERIT, FALSE );
SpecialsWindow:Special1Object = CardAuthorization;
SetSlotOption( SpecialsWindow:Special1Object, IF_NEEDED, NULL );
SetSlotOption( SpecialsWindow:Special1Object, WHEN_ACCESS, NULL );
SetSlotOption( SpecialsWindow:Special1Object, BEFORE_CHANGE, NULL );
SetSlotOption( SpecialsWindow:Special1Object, AFTER_CHANGE, NULL );
SetSlotOption( SpecialsWindow:Special1Object, IMAGE, Special1ObjectListBox );
MakeSlot( SpecialsWindow:Special2Object );
MakeSlot( SpecialsWindow:Test1 );
SetSlotOption( SpecialsWindow:Test1, INHERIT, FALSE );
SpecialsWindow:Test1 = InternalSubsystem;
SetSlotOption( SpecialsWindow:Test1, IF_NEEDED, NULL );
SetSlotOption( SpecialsWindow:Test1, WHEN_ACCESS, NULL );
SetSlotOption( SpecialsWindow:Test1, BEFORE_CHANGE, NULL );
SetSlotOption( SpecialsWindow:Test1, AFTER_CHANGE, NULL );
MakeSlot( SpecialsWindow:Test2 );
SetSlotOption( SpecialsWindow:Test2, INHERIT, FALSE );
SpecialsWindow:Test2 = ExternalSubsystem;
```

```
SetSlotOption( SpecialsWindow:Test2, IF_NEEDED, NULL );
SetSlotOption( SpecialsWindow:Test2, WHEN_ACCESS, NULL );
SetSlotOption( SpecialsWindow:Test2, BEFORE_CHANGE, NULL );
SetSlotOption( SpecialsWindow:Test2, AFTER_CHANGE, NULL );
SetValue( SpecialsWindow:BackgroundColor, 0, 0, 255 );
SpecialsWindow:Menu = FALSE;
SpecialsWindow:Titlebar = TRUE;
SpecialsWindow:Sizebox = TRUE;
 ResetWindow ( SpecialsWindow );



/*******************************************************/
/**       ALL RULES ARE SAVED BELOW           **/
/*******************************************************/

    /***********************************
    **** RULE: RuleTest1
    ***********************************/
MakeRule( RuleTest1, [],
   GetNthElem( GetValue( ObjectClass, Global:Object ), 1 )
      #= MainMiniComputer And GetNthElem( GetValue( ObjectClass,
                                 Global:Object ),
                        2 ) #= CentralMiniMainframe,
   Let [object Global:Object]
      SetNthElem( StepSD1:object, 1, Test1 ) );

    /***********************************
    **** RULE: RuleTest2
    ***********************************/
MakeRule( RuleTest2, [],
   GetNthElem( GetValue( ObjectClass, Global:Object ), 1 )
      #= MainMiniComputer And GetNthElem( GetValue( ObjectClass,
                                 Global:Object ),
                        2 ) #= ExternalMiniMainframe,
   Let [object Global:Object]
      SetNthElem( StepSD1:object, 2, Test2 ) );

    /***********************************
    **** RULE: RuleTest3
    ***********************************/
MakeRule( RuleTest3, [],
   GetNthElem( GetValue( ObjectClass, Global:Object ), 1 )
      #= SpecialComponent And Not( GetNthElem( GetValue( ObjectClass,
                                    Global:Object ),
                           3 ) #= None ),
   Let [object Global:Object]
      SetNthElem( StepSD1:object, 3, GetNthElem( ObjectClass:object,
                                  3 ) ) );

      /***********************************
      **** RULE: Rule4
      ***********************************/
MakeRule( Rule4, [],
   GetNthElem( GetValue( StepSD1, Global:Object ), 1 )
      #= Test1,
   Let [object Global:Object]
      Let [newslot GetNthElem( GetValue( StepSD1, object ), 1 )]
         {
         If Not( Slot?( StepSD1:newslot ) )
           Then {
              MakeSlot( StepSD1:newslot );
              SetSlotOption( StepSD1:newslot, MULTIPLE );
              AppendToList( StepSD1:Test1, newslot );
              };
         AppendToList( StepSD1:newslot, object );
         } );

      /***********************************
      **** RULE: Rule6
      ***********************************/
MakeRule( Rule6, [],
   Not( GetNthElem( GetValue( StepSD1, Global:Object ), 3 )
        #= None ),
   Let [object Global:Object]
      Let [newslot GetNthElem( GetValue( StepSD1, object ), 3 )
                # SubSystem]
         {
         If Not( Slot?( StepSD1:newslot ) )
           Then {
              MakeSlot( StepSD1:newslot );
              SetSlotOption( StepSD1:newslot, MULTIPLE );
              AppendToList( StepSD1:Specials, newslot );
              };
         AppendToList( StepSD1:newslot, object );
         } );

      /***********************************
      **** RULE: Rule5
      ***********************************/
MakeRule( Rule5, [],
   GetNthElem( GetValue( StepSD1, Global:Object ), 2 )
```

```
    #= Test2,
Let [object Global:Object]
   Let [newslot GetNthElem( GetValue( StepSD1, object ), 2 )]
      {
      If Not( Slot?( StepSD1:newslot ) )
        Then {
           MakeSlot( StepSD1:newslot );
           SetSlotOption( StepSD1:newslot, MULTIPLE );
           AppendToList( StepSD1:Test2, newslot );
           };
      AppendToList( StepSD1:newslot, object );
      } );
```

INDEX