# A Methodology for the High Level Design Of Object-Oriented Software

by

Said Abdallah Muhammad

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

**MASTER OF SCIENCE**

In

**COMPUTER SCIENCE**

January, 1993

# A methodology for the high level design of object-oriented software

Muhammad, Said Abdallah, M.S.

King Fahd University of Petroleum and Minerals (Saudi Arabia), 1993

# A METHODOLOGY FOR THE HIGH LEVEL DESIGN OF OBJECT-ORIENTED SOFTWARE

BY

SAID ABDALLAH MUHAMMAD

A Thesis Presented to the

FACULTY OF THE COLLEGE OF GRADUATE STUDIES

KING FAHD UNIVERSITY OF PETROLEUM & MINERALS

DHAHRAN, SAUDI ARABIA

In Partial Fulfillment of the
Requirements for the Degree of

**MASTER OF SCIENCE**

**In**

COMPUTER SCIENCE

JANUARY 1993

**KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS**
**DHAHRAN, SAUDI ARABIA**

**COLLEGE OF GRADUATE STUDIES**

This thesis, written by **SAID ABDALLAH MUHAMMAD** under the direction of his Thesis Advisor and approved by his Thesis Committee, has been presented to and accepted by the Dean of the College of Graduate Studies, in partial fulfillment of the requirements for the degree of **MASTER OF SCIENCE** in **COMPUTER SCIENCE.**

Thesis Committee

Muhammad Shafiq

Dr. MUHAMMAD SHAFIQUE
Thesis Advisor

Bassel Arafeh

Dr. BASSEL ARAFEH
Member

[illegible]

Dr. MARWAN AL-AHMADI
Member

S. Ghanta

Dr. SUBBARAO GHANTA
Member

[illegible]

Dr. MUHAMMAD AL-MULHEM
Chairman, Information and Computer Science Department

[illegible]

Dr. ALA H. AL-RABEH
Dean, College of Graduate Studies

Date: 30-1-1993

عمادة الدراسات العليا
COLLEGE OF G

**Dedicated to Sakina and Abdullatif**

## ACKNOWLEDGEMENTS

Acknowledgement is due to King Fahd University of Petroleum and Minerals for supporting this research.

I would like to express my deep gratitude to Dr. Muhammad Shafique, my thesis advisor, for his guidance. I am also grateful to Dr. Bassel Arafeh, Dr. Marwan Al-Ahmadi, and Dr. Subbarao Ghanta, the other members of my thesis committee, for their help and numerous suggestions.

I am indebted to Dr. Muhammad Al-Mulhem, the Chairman of the Department of Information and Computer Science, King Fahd University of Petroleum and Minerals, and Dr. Muhammad Al-Tayyeb, former Chairman of the Department, for their assistance and support.

I would like to thank Mr. Husni Al-Muhtaseb for translating the abstract of this thesis into Arabic, and Mr. Abdallah Abbas for typing the Arabic abstract.

Thanks go to Mr. Athanase Kanamugire and Mr. Shoaib Qureshi of the Main Library, King Fahd University of Petroleum and Minerals, for helping me in my quest for references.

Thanks to all my colleagues at King Fahd University of Petroleum and Minerals for their encouragement.

Finally, I express my appreciation and gratitude to my wife for her encouragement, patience, sacrifice, and understanding while this work was in progress.

# TABLE OF CONTENTS

Page

Page

Page

Page

# LIST OF FIGURES

Figure Page

## THESIS ABSTRACT

| | |
|---|---|
| **NAME OF STUDENT** | : SAID ABDALLAH MUHAMMAD |
| **TITLE OF STUDY** | : A METHODOLOGY FOR THE HIGH LEVEL DESIGN OF OBJECT ORIENTED SOFTWARE |
| **MAJOR FIELD** | : COMPUTER SCIENCE |
| **DATE OF DEGREE** | : JANUARY,1993 |

The object-oriented approach to programming and designing software systems is receiving tremendous attention in the programming languages, knowledge representation, and database disciplines. The design of a software system determines its major characteristics and has great influence on later phases of the software life cycle, particularly the maintenance phase, which accounts for most of the cost of software system over its entire life cycle.

Object-orientation offers a new design paradigm which can help to manage the growing complexity and increasing costs of software development and maintenance. The paradigm also promises greater productivity by facilitating the building of reliable applications from reusable components. However, proposed object-oriented design methodologies are still at an initial stage of investigation. Most of these methodologies do not incorporate the dynamic model; the model which captures the dynamic behaviour of a software, and almost all of them do not consider design-consistency checking in greater detail at the high-level design stage.

We develop an object-oriented, high-level, software design methodology. The methodology is based on two models: a static model, and a dynamic model. The static model uses a form of an enhanced entity-relationship model, inheritance diagrams, and CRC (Class Responsibility Collaboration) cards to capture the static structure of the classes and class relationships. The dynamic model uses a modified Harel's Statechart notation to capture the internal behaviour of each class and the message passing behaviour among collaborating classes in the system. We also develop a number of design-consistency checking algorithms.

**MASTER OF SCIENCE DEGREE**
**KING FAHD UNIVERSITY OF PETROLEUM AND MINERALS**
**Dhahran, Saudi Arabia**
**JANUARY,1993**

# ملخص الرسالة

| | |
|---|---|
| اسم الطالـــب : | سعيد عبدالله محمد |
| عنوان الرسالة : | طريقه للتصميم العام للأنظمة المعتمدة على الكائنات . |
| التخصــــص : | علوم الحاسب |
| تاريخ الدرجة : | كانون ثاني ( يناير ) ١٩٩٣ م |

يلاحظ حالياً ازدياد الاهتمام ببناء وتصميم البرامج المعتمدة على طريقة الكائنات في مجال تمثيل المعرفة وقواعد البيانات ولغات البرمجة . ومن المعلوم ان مرحلة تصميم نظام ما ، تحدد صفات النظام كما تؤثر تأثيراً كبيراً على المراحل الاخرى التي تمر بها عملية بناء النظام وخاصة مرحلة الصيانة التي عادة ما تكون تكلفتها اعلى من تكلفة اية مرحلة اخرى .

يوفر استخدام الكائنات طريقة جديدة للتصميم تساعد على تقليل التعقيد والتكلفة اللازمة لتطوير النظم وصيانتها . كما تشير الطريقة الى امكانية تحسين الانتاجية وذلك ببناء تطبيقات جديدة من عناصر قابلة لإعادة الاستخدام . ومع كل هذا ، فما زالت طرق التصميم المعتمدة على الكائنات في بداية طريقها . واغلب هذه الطرق لا تأخذ باعتبارها النموذج المتغير الذي يمثل طبيعة التغير في النظام. كما ان اغلب هذه الطرق لا تأخذ باعتبارها فحص توافق التصميم العام بالتفصيل المطلوب .

يقدم هذ البحث طريقة للتصميم للأنظمة التي تعتمد على الكائنات . وتعتمد الطريقة المقترحة على نموذجين : الاول هو النموذج الثابت والثاني هو النموذج المتغير . ويستعمل النموذج المتغير شكلاً مطوراً من اشكال نموذج علاقة الاشياء ورسومات التوراث اضافة الى مسؤولية الاصناف وارتباطاتها كي تشمل النموذج الثابت للاصناف وعلاقتها ببعضها .

وقد استخدمنا في النموذج المتغير رموز هاريل للرسم ليشمل السلوك الداخلي لكل صنف ووضع الرسائل بين الاصناف المتفاعلة في النظام .

كما تم تطوير عدد من خوارزميات فحص توافق – التصميم .

ماجستير في العلوم

جامعة الملك فهد للبترول والمعادن

الظهران – المملكة العربية السعودية

كانوني ثاني ( يناير ) ١٩٩٣ م

# CHAPTER 1
# INTRODUCTION

## 1.1 Software design: An overview

### 1.1.1 Definition

Software design is a creative, non-algorithmic problem solving process involving tradeoffs. It is a process of conceptualising and generating modules of a software product with their design alternatives, evaluating these design alternatives to eliminate unacceptable ones, and then iteratively refining and integrating the surviving alternatives into more detailed design until the results satisfy user requirements within the allowed constraints.

### 1.1.2 Classification of design methodologies

Most of the design methodologies developed can be classified as either top-down functional design, data-driven design, or object-oriented design. In top-down functional design, the system is viewed in terms of its functions. Starting at a high-level view, each level is refined into successively more detailed levels of functions. In data driven design, the structure of the data processed by the system is used to derive the structure of the software. Object-oriented design is based on the use of data and procedural abstraction. Each data and procedural abstraction constitutes an object, and the system is composed of a collection of objects.

### 1.1.3 Important Software development lifecycle models

#### The iterative waterfall model

The iterative waterfall model consists of five phases : (1) requirements analysis and specification, (2) design, (3) implementation, (4) testing, and (5) operation and maintenance. There is much iteration and feedback among the steps and phases of the lifecycle. The requirements analysis and specification phase includes user-needs analysis and feasibility study. The design phase covers user-interface design, preliminary design, and detailed design. The implementation phase involves translation of detailed design specifications into a source code. This code is then compiled and executed . If any errors are discovered, debugging and re-coding takes place. Testing occurs at three levels : unit testing, system testing, and acceptance testing. Following acceptance by the user the software system is released for operation and it enters the maintenance phase. There are four types of maintenance activities : corrective maintenance, adaptive maintenance, perfective maintenance, and preventive maintenance.

#### The prototyping model

In the prototyping model, the requirements specifications are identified, perhaps in a preliminary way if they are unclear or fuzzy. They are then implemented in the form of a prototype as a part of a prototyping process, and then they are fed back to the user for validation. The sequence of events is then to: (1) design a prototype, (2) implement the prototype, (3) submit the results of this implementation to the user for feedback as to the validity of the design

(4) refine the requirements specifications and the prototype on the basis of the feedback, and (5) repeat the process until there is satisfaction with respect to the results obtained.

**The spiral model**

The spiral model [Boe86] requires the use of prototyping. It proceeds through an iterative sequencing of the several phases of the model for each time a different prototype is developed. The model includes an evaluation of risk before proceeding to each subsequent elaboration of phases in the development of a software product. The opportunity to assess the various risk factors such as cost, hardware requirements, and performance, permits the developer to determine whether to continue with the development, seek an alternative path, or abort the process [SaP90].

**The knowledge-based lifecycle model**

The knowledge-based software lifecycle model is based on the use of a particular lifecycle model with the application of expert systems. Knowledge-based models have been developed primarily to assist the software designer realise a specific design through the application of rule based systems [SaP90].

### 1.1.4 The Requirements analysis and specification phase

Requirements analysis involves the identification and development of user requirements. Software specifications are the translation of these user requirements to technical specifications for the software system that will be developed. The goal of specification is to completely specify technical requirements for the system, including the precise definition of what the functions of the software are, the constraints under which it will perform its functions, default and error conditions and the acceptable responses to these conditions.

There are two categories of system and software requirements specifications : functional and non-functional. Functional requirements are those that relate directly to the functioning of the system. They describe how the system should behave given certain input. Non-functional requirements are requirements or restrictions posed by the user or the problem that do not relate to the functions or operations to be performed by the system [Myn90]. The major non-functional requirements are associated with quality factors, user interfaces, performance characteristics, operating constraints, economic constraints, and political constraints. The commonly used quality factors are : efficiency, flexibility, integrity, inter-operability, maintainability, portability, reliability, responsiveness, reusability, testability, understandability, and usability. Since certain of these quality factors conflict in specific development situations, priorities must be assigned to the particular quality factor characteristics required.

### 1.1.5 The Design phase

The design phase consists of three sub-phases : user-interface design, preliminary design, and detailed design. The user-interface design can often be carried out in parallel with the preliminary design.

The inputs to the preliminary design phase are the detailed software specifications that were formulated in the requirements analysis and specifications phase. The goal of preliminary design is to specify a software architecture that satisfies the requirements, the user-interface design specifications, and the implementation constraints. The system architecture is specified in terms of a hierarchically structured collection of modules, each of which plays some well defined part of the systems capabilities. In addition to the delineation of the modules, their interfaces and interactions are defined in order to specify the coordination needed to assure the achievement of the expected functional capabilities.

The architectural structure defined in the preliminary design phase is refined into implementation details during detailed design. Data structures are defined, algorithmic details are specified, decisions are made concerning how to implement each of the components specified in the preliminary design, and the test plan is prepared.

### 1.1.6 The quality factors of a design methodology

The following quality factors can be used to judge the quality of a design methodology: (a) relevance, (b) usability, (c) reliability, (d) efficiency, (e) maintainability, and ( f) robustness.

**(a) Relevance**

Relevance is a measure of the degree to which a design methodology meets the need of designers in designing a particular class of software products.

**(b) Usability**

The usability of a design methodology is a measure of the effort and prerequisite knowledge required in order to learn, understand and use the methodology

**(c) Reliability**

The reliability of a design methodology is the probability that the methodology is as effective as it is claimed to be. The reliability of a methodology depends on its supporting tools and its underlying design principles. A methodology cannot be reliable without a comprehensive set of design tools, nor can it be reliable if it is not based on the following design principles :

(i) **The principle of abstraction** (i.e. concentrating on a problem at some level of generalisation without regard to low-level details ). This embraces :

(a) **The principle of evolution** : Design consists of a large number of relatively simple steps; at each step more details are incorporated into the solution.

(b) **The principle of separation of concerns** : No more than one aspect of design is attempted at a time.

(c) **The principle of implementation-independent design** : Decisions on details of implementation are delayed as long as practical.

(ii) **The principle of modularity** : The software is organised into modules, each of which encapsulates a specific set of abstract capabilities.

(iii) **The principle of perceivability** : Every work-product of the design is sufficiently simple.

(iv) **The principle of completeness and unambiguity** : Every work-product of the design process is complete and unambiguous.

(v) **The principle of user-centered design** : Every design decision gives priority to the needs of the users.

**(d) Efficiency**

The efficiency of a design methodology is a measure of its capability to improve productivity of the design process and to ensure that design products are of specified quality. The key methods for productivity improvements are fault avoidance and fault removal methods. These methods are especially cost-effective when applied to the early stages of the software development process [Law88]. The effectiveness of a methodology in fault avoidance can be assessed by examining ways in which the methodology :

- keeps design task complexity under control,
- controls complexity of work-products,
- ensures that design tasks and the resulting work-products are completely and unambiguously defined,
- ensures that all documentation is complete and unambiguous,
- Leads to the identification of " off the shelf " parts,
- supports the use of automated aids for software design.

The effectiveness of a methodology in fault removal may be assessed by examining :

- procedures for inspection of work-products,

- procedures for initiating, performing and documenting modifications .

**(e) Maintainability**

Maintainability is a measure of the effort required to keep the methodology itself up to date to suit changing requirements.

**(f) Robustness**

Robustness can be regarded as a measure of the resistance of the methodology for misuse. Features which help make a methodology robust are :

- clear specification of intermediate work-products and deliverables,
- clear sequencing of design tasks,
- specified checkpoints and reviews.

## 1.2 Motivation for the study

- Current Object-Oriented design methodologies do not consider design consistency in detail at the high-level design stage.
- Most Object-Oriented design methodologies incorporate static models without dynamic models. The static model captures the structural and semantics relationship among the classes comprising a software. The dynamic model captures the dynamic behaviour of these classes. Design methodologies without dynamic models include those proposed by Coad and Yourdon [CoY91], Meyer [Mey88], WirfsBrock et al. [WWW90], and Shlaer and Mellor [ShM88].
- The dynamic and static models are not well integrated in Object-Oriented design methodologies with dynamic models. Design methodologies with

dynamic models include those proposed by Alabiso [Ala88], Booch [Boo91], and Rumbaugh et al. [RBP+91].

## 1.3 Objectives for the study

- Develop an Object-Oriented Design methodology for the high-level design stage incorporating design-consistency checking.
- Develop a dynamic-model for the design methodology which will easily integrate with the static-model.
- Study aspects of high-level object-oriented design which can be fully or partially automated.

## 1.4 Thesis organisation

In Chapter 2 we present a literature survey which covers concepts of object-orientation, the benefits of object-oriented design, and current object-oriented design methodologies. The semantic notations used in the static model of the developed design methodology are given in Chapter 3. The developed design methodology is presented in Chapter 4. In Chapter 5 we discuss in detail design-consistency checking algorithms as applicable at the high-level design stage. In Chapter 6 we elaborate the design methodology in a case study of an Automated Teller Machine network. Finally, in Chapter 7, conclusions and recommendations for future work are presented.

# CHAPTER 2

# LITERATURE SURVEY

## 2.1 Concepts of Object-Orientation

### 2.1.1 OOD: A new design paradigm

Object-oriented design is an approach to software design where the system is designed as a set of objects which model problem domain entities and which interact through message passing. Each object encapsulates its own private state (instance variables) and methods (procedures and/ functions) which, alone, can act on that state.

Historically, software development has progressed from a purely procedural approach to a data driven approach and now to the object-oriented approach. The progression has resulted from a gradual shift in point of view of the development process. The procedural design paradigm utilises functional decomposition to specify the tasks to be completed in order to solve a problem. The data-driven approach gives more attention to data specifications than the procedural approach but still utilises functional decomposition to develop the architecture of the system. These two approaches are based around the notion of representing the system state in a centralised shared memory which may be accessed by the functions/ procedures. As these functions/procedures execute, the system state is updated. An object-oriented approach to design is different from the procedural and data driven approaches in two principal ways:

1) The basic abstractions are not real-world functions, but real world entities,
2) State information is not represented in a centralised shared memory, but is distributed amongst the objects modelling the system.

### 2.1.2 Behaviour sharing in Object-Oriented Languages

Object-oriented programming languages can be classified as class-based or object based according to their behaviour sharing mechanisms. C++ [ElS90], Eifell [Mey88], Smalltalk-80 [GoR83], and CommonObjects [Sny86] are examples of class-based languages. Actors [Agh86], ABCL/1 [YBS86], [Yon90], and Self [UnS87] are examples of object-based languages.

#### (a) Class-based behaviour sharing

In class-based languages a class denotes a set of similar but unique objects. All objects belonging to the same class are described by the same instance variables and the same methods.

#### Inheritance

Inheritance is a structural relationship (often called the " is-a" relationship) between classes which allows a class to inherit methods and instance variables from ancestor classes (superclasses) , and to have its methods inherited by descendant classes (subclasses). A subclass may add new methods and new instance variables to that of its superclass. A local definition may mask or override an inherited definition.

Multiple inheritance occurs when a class inherits from more than one direct superclass. Problems arise with multiple inheritance when a class tries to inherit from two or more superclasses that contain methods and/or instance variables having similar names. Solutions have ranged from regarding name conflicts as errors which must be disambiguated to allowing the programmer full control over how multiple inherited methods are invoked.

CommonObjects [Sny86] and Sina/st [Akt88] provide selective inheritance, a notion not supported by most object-oriented languages.

**Polymorphism**

Polymorphism is the ability of an entity, such as a variable or function argument, to refer at run-time to instances of various classes. Hence, the actual operation performed on receipt of a message depends on the class of the instance.

Polymorphism is implemented in different ways according to whether the language used is dynamically-typed or statically-typed. In dynamically-typed languages such as Smalltalk-80 [GoR83], Objective-C [Cox86], CLOS [Moo89], and Flavors [Moo86] entities have no static types, so that they may at run- time refer to objects of any class; when an operation is requested on an entity, its dynamic state determines what realisation, if any, is available for the operation.

In statically-typed languages such as C++ [ElS90], Eiffel [Mey88], Hybrid [Nie87], and Object Pascal [Sch86] polymorphism occurs through the difference between the declared (static) class of a variable and the actual (dynamic) class of the value contained in the variable. This difference is maintained within the framework of the is-a relationship. A variable can hold a value of the same type as that of the declared class of the variable, or any subclass of the declared class.

A polymorphic method is one that may accept arguments of different types. In universal polymorphism, there is one implementation of a method that works for all types (parametric polymorphism) or all subsets of a type (inclusion polymorphism).

Ad-hoc polymorphism is realised through overloading or through overloading and coercion of arguments.

**Abstract Class**

An abstract class is a class that is not intended to produce instances of itself. It exists so that behaviour common to one or more classes can be factored out into one common location.

**Deferred Class**

A deferred class is an abstract class that contains one or more deferred methods. The class defines the specification (signature) of a deferred method but not the implementation. Various implementations are defined in the subclasses of the deferred class. The implementation used depends on the dynamic type of the deferred class which will always be one of its subclasses. Languages which support deferred classes include C++ [EIS90] and Eiffel [Mey88].

We call a deferred class in which all the methods are deferred as a complete deferred class; otherwise we call it a partially deferred class.

**Parameterised Class**

A parameterised class is a class which contains generic parameters in its definition. A parameterised class is instantiated to specific classes when actual type parameters are supplied. Languages which support both parameterised classes and inheritance include Eiffel [Mey88] and Trellis/Owl [SCB+86]. Parametrised classes have been proposed as an extension to C++ [ElS90].

**Metaclass**

Some Object-Oriented programming languages support the view that classes are objects. The notion of a metaclass as a class of a class is then used. A metaclass is itself an object, and it is an instance of yet a higher level metaclass. The arbitrary recursion must be broken by an arbitrary root metaclass. The concept of metaclasses permits the specialisation of behaviour for the initialisation of individual classes without leaving the object-oriented framework. The argument against using this concept is that it complicates the object-oriented paradigm. Systems supporting explicit metaclassess include LOOPS [Bob83], ObjVlisp [Coi87], and CLOS [Moo89],[Kee89]. Smalltalk-80 [GoR83], ABCL/1 [Yon90], Objective-C [Cox86] support implicit metaclasses; when a class is defined, a new metaclass is automatically created by the system.

**(b) Object-based behaviour sharing**

Delegation [Lie86], [Lie87] is presented as an approach to sharing behaviour in object-based languages. In delegation an object knows about one or more objects called proxies or prototypes. At run-time the object will delegate to its proxies a message that it does not know how to handle. These proxies can in turn delegate the message to their own proxies. A proxy which can handle the message will handle it in place of the initial receiver of the message.

There is an ongoing debate on the relative merits of class inheritance as opposed to those of delegation in object-oriented programming [MiR89], [Lie86], [Ste87], [TSK90].

### 2.1.3 Composite Class

A composite class is a class defined as a composition or an aggregation of other classes called its parts or components. Some knowledge representation languages such as LOOPS [Bob83], YAFOOL [Duc90] and OBJLOG [Dug91] provide features for defining such classes. In these languages the creation of a composite object leads to the creation of an instance of each component [MNC+91].

## 2.2 Why OOD ?

OOD is a new design paradigm which has the following advantages over traditional function-oriented design paradigms:

**(a) Understandability**

The Object-oriented approach helps to manage the complexity of software development by modelling the real-world in terms of its objects. Objects encapsulate both data and state; they may be considered as stand-alone entities without reference to other parts of the system and thus they are easily understood. Because of this, errors of omission and errors due to misunderstood abstractions are more obvious and more likely to be detected by the designer before the design is complete. In a function- oriented design, the designer must separate state information from functions and in doing so he may leave out either essential state variables, required functions or both.

**(b) Reusability**

By removing the need for dealing with shared state and by removing the interdependency among software components, OOD permits the development of reusable software components. The major approaches facilitating code reuse are polymorphism, behaviour sharing, and parameterised classes.

A polymorphic component can have several types; hence it can be used in several different ways. When behaviour is inherited from another class or when an object can refer the implementation of behaviour to other objects, the code that provides that behaviour does not have to be rewritten. Some object-oriented languages support parameterised types. This provides a different style of code reuse by describing generic components.

Several libraries of reusable software components have become commercially available : The Smalltalk class hierarchy [GoR89], the McApp classes for graphical user-interface design [WRS90], the ICpak 201 collection of Objective-C classes [Kno89], the National Institute of Health C++ collection [GLP90], the InterViews collection of C++ classes for user-interface design [LVC89], and the Eiffel Libraries [Mey90].

The results of using reusable components to construct software systems are: faster software development time, decreased maintenance costs, and increased software reliability.

**(c) Modifiability**

Understandability, the absence of shared data area, and the loose coupling of software components makes it easy to modify an object-oriented application. Changes to one part of the design are less likely to affect other design components. New types of objects can be added without drastically changing the existing structure of the application; inheritance coupled with polymorphism minimises the amount of existing code that must be changed when extending a system. In a function-oriented approach changes to one function often require changing state information .This can have unanticipated side effects on other parts of the system.

**(d) Fault repair**

Understandability, the absence of shared data area, and the loose coupling of software components make it easy to repair a fault in an object-oriented application. Fault repair in one part of the system is unlikely to introduce new faults elsewhere in the system.

**(e) Fault tolerance**

The distribution of state information which is inherent in an object-oriented approach helps to make an object-oriented application fault tolerant. In a situation where all of the system state is accessible by the functions manipulating that state, an error in one of these functions can be readily propagated through the entire system state. This is likely to cause complete system failure. However, where only part of the state is visible to an object at any time, only the state within that object may be damaged in the event of an error. Damage is not likely to be propagated throughout the system, thus improving the overall probability that the system can continue to operate in the presence of an error.

**(f) Reliability**

Reusability, modifiability, fault repair, and fault tolerance all contribute to reliability. The more situations in which code has been reused, the greater will have been the opportunity of discovering errors.

## 2.3 Current OOD methodologies

### 2.3.1 The SA/OOD controversy

Object-oriented design methodologies are still in their early stages; no standard methodologies have been established. Thus, current methodologies are varied. Some meld the top-down approach of structured analysis (SA) and object-oriented design, others use a unified object-oriented paradigm throughout all design activities. There is an ongoing controversy as to whether structured analysis is compatible with object-oriented-design. Some methodologists advocate that they are compatible [Ala88], [WaM89], [Shu91] and others advocate that they are incompatible [Bai89], [ArY90], [CoY90], [Bro91], [Fir91].

## 2.3.2 Ada-based object-oriented design methodologies

What has come to be known as object-oriented-design in the context of Ada was first proposed by Abbot [Abb83], and later formalised and extended by Booch [Boo83], [Boo86]. Booch's original work on object-oriented-design was strongly influenced by the characteristics of Ada packages and tasks, he has now extended that work to include more general class relationships as they exist in object-oriented programming languages [Boo91].

The Object-oriented design methodology proposed by Booch consists of three steps : 1) Define the problem, 2) Develop an informal strategy, and 3) Formalise the strategy. The third step has four sub-steps : 3.1) Identify the objects and their attributes, 3.2) Identify the operations suffered by and required of each object, 3.3) Establish the interface of each object, 3.4) Implement the operations in each object. Originally Booch's methodology derived a design from textual specification, later on he changed this to a data flow diagram specification. Booch uses a diagramming notation that shows the dependencies between Ada packages and tasks which implement the objects.

Jalote [Jal89] proposed an extended object-oriented design methodology which incorporates a top-down, step-wise refinement process consisting of a functional and an object refinement phase. Object refinement allows composite entities in real-life to be modelled in terms of nested objects. The functional refinement phase iteratively refines the operations that do not seem to belong to any particular object uncovered so far in the refinement process. The iteration stops when no operation for further refinement are identified.

GOOD (General Object- Oriented Software Development) was developed by Seidewitz and Stark [SeS86], [Sei89]. GOOD addresses the requirements specification and design phases of an Ada-oriented software development life-cycle. Structured analysis is used to develop the specification. Abstraction analysis is then used to make a transition to an object-oriented design by recursively producing object diagrams. Object-diagrams are used to represent two orthogonal hierarchies : composition hierarchy and seniority hierarchy. The first step in abstraction analysis is to find the central object diagram of the system. This central object diagram is then recursively refined into lower level diagrams, forming a levelled seniority hierarchy of object-diagrams. This process is continued until all the processes and data stores are associated with object diagrams. When all the object diagrams are found, the seniority hierarchy is recursively collapsed by distributing control among the object diagrams in each level. In doing this, data flow arrows are removed by object descriptions. An object description includes a list of all operations provided by an object and a list of operations provided from other objects.

HOOD (Hierarchical Object- Oriented Design) [WPM89], [WPM90], [HOH91] is closely related to Booch's work. It combines two complementary methods : Abstract machines and Object-Oriented design. Like Booch's approach, it starts by decomposing the problem into objects from the nouns and verbs used in a textual description. The system is described as a seniority hierarchy of objects each of which is either active or passive. A passive object provides facilities, whereas an active object not only has a hierarchical relationship with its sub-objects, but also controls the order of their execution. An object can access the visible facilities of another with the use relationship. HOOD supports concurrency, it also addresses other real-time facilities such as exception handling and time-outs. An important goal of HOOD is to map its features directly to Ada concepts through an Ada-like Program Design Language.

MOOD (Multiple-view Object-Oriented Design Methodology) [WPM89] is a method for structured object-oriented design. It supports the construction of programs from an analysis model developed with Ward / Mellor's Structured Analysis with Real-time extensions [WaM85]. The method supports the object-oriented paradigm, but allows concurrent processes to be expressed as tasks rather than objects. MOOD addresses different issues, including the identification of objects and tasks, how objects and tasks influence other objects and tasks, and sequential execution of routines.

PAMELA (Process Abstraction Method for Embedded Large Applications) [Sei89], [BuW90] is an Ada-specific design method for real-time and embedded systems devised by Cherry in 1986. The method uses a powerful graphical notation. In the PAMELA design process, the designer successively decomposes processes into concurrent sub-processes until he reaches the level of primitive, single-thread processes. The relationships between the processes is represented by data flows.

OOSD (Object-Oriented System Design) is a graphical design notation developed by Wasserman et al. [WPM89], [WPM90]. It provides a hybrid notation for combining Structured Analysis and Object-oriented techniques. The notation is based on structure-charts from Structured Analysis. Booch's notation for Ada packages and tasks, class hierarchy and inheritance principles from object-oriented programming, and Hoare's monitors. OOSD provides notation for message arguments, classes, generic classes, inheritance, polymorphism, exceptions, visibility relationship, and asynchronous processes. The goal of OOSD is to provide a single architectural design that can support every software design [WPM89]. OOSD does not address the method by which a design would be derived. It is expected that designers will develop and use their own design methods within OOSD's framework.

Walters [Wal91] describes an Ada object-based analysis and design approach consisting of : object identification and characterisation, real-time behaviour specifications for object services, verification that the object model satisfies top-level requirements, definition of Ada constructs from object characterisations, and design refinement and validation.

**Limitations of the Ada-based OOD methodologies**

The close coupling of the Ada-based methodologies to Ada results in some significant limitations in designing software systems by the methodologies. Major features of the object-oriented paradigm such as classes, abstract classes, inheritance, polymorphism, and message passing are not addressed directly in the analysis and design notations. OOSD is an attempt to redress this shortcoming, but the very idea of having a hybrid notation that can support every software design is debatable. Wasserman et al. [WPM89] admit that such a hybrid notation might not be aesthetically appealing.

### 2.3.3 Object-Oriented Analysis methods

Gibson [Gib90] describes an object-oriented approach to analysis called OBA (Object Behaviour Analysis). OBA facilitates an initial understanding of an application in terms of behaviours and objects. It then specifies how to analyse the application, with the aim of producing a requirements specification. The approach encourages iteration within and between the different phases of development. It has five steps : Identifying the behaviours of the system, deriving objects using the behavioural perspective, classifying the objects, identifying relationships among objects, and modelling processes.

Rubin and Goldberg [RuG92] have developed an object-oriented analysis methodology also called Object Behaviour Analysis (OBA). The methodology consists of five steps: setting the context of the analysis, understanding the problem by focusing on behaviours, defining objects that exhibit behaviours, classifying objects and identifying their relationships, and modelling system dynamics.

Nerson [Ner92] describe an object-oriented analysis and design methodology based on a notation called BON (Better Object Notation). The methodology consists of the following nine steps: delineate the system borderline, list candidate classes observed in the problem domain, group classes into clusters, define candidate classes in terms of commands/constraints, define behaviours, define class features, invariants and contracting conditions, refine class descriptions, work on generalisation, complete and review the architecture.

Coad and Yourdon [CoY90] have set forth an object-oriented analysis model consisting of five layers : finding classes and objects, identifying structures, identifying subjects, defining attributes, and defining services.

Shlaer and Mellor [ShM88] have developed an object-oriented analysis method consisting of six steps: develop an information model, define object life-cycles, define the dynamics of relationships, define system dynamics, develop process models, and define domains and subsystems

.

Hayes and Coleman [HaC91] present a set of formally based coherent models for object-oriented analysis. The models extend current informal object-oriented analysis techniques. They have precise semantics and they constitute a consistent description technique for domain analysis.

Dennis de Champeaux [Den91] outlines a top-down object-oriented analysis method consisting of three steps : developing an information model, developing a state transition model, and developing a process model.

### 2.3.4 Metrics for Object-Oriented Design

Chidamber and Kemerer [ChK91] propose six software metrics for object-oriented design as a first attempt at developing formal metrics for OOD. They then evaluate the proposed metrics against Weyuker's list of software metric evaluation criteria [Wey88] and provide the formal results of that evaluation. The proposed metrics are : Weighted Methods Per Class (WWC), Depth of Inheritance Tree (DIT), Number of Children of a class (NOC), Coupling between

Objects (CBO), Response set For a Class (RFC), and Lack of Cohesion in Methods (LCOM).

### 2.3.5 Pure Object-Oriented Design Methodologies

Coad and Yourdon [CoY91] define an object-oriented design model consisting of five layers : subject layer, class and object layer, structure layer, attributes layer, and service layer. These five layers correspond to the five layers in their OOA model [CoY90].

Gossain and Anderson [GoA90] describes an iterative design approach for designing reusable object-oriented software. The approach concentrates on designing code that is to be part of a hierarchy or framework of classes for an application domain. It has five stages : Identification of initial candidate classes by domain analysis, the creation of abstract classes from the initial initial candidate classes, the derivation of the concrete classes of the domain, the fine tuning of classes, and coding.

Wirfs-Brock et al. [WiW89], [WWW90] have developed a responsibility-driven design methodology based on the client / server model. A server provides a set of services to a client upon request. The methodology has two phases : an initial exploratory phase and a detailed analysis phase. The initial exploratory phase is subdivided into : finding the classes in the system, determining what operations each class is responsible for and what knowledge it should maintain, and determining the ways in which objects collaborate with other objects to discharge their responsibilities. The detailed analysis phase consists of : factoring common responsibilities in order to build class hierarchies, and streamlining the collaboration between classes.

OORASS (Object-Oriented Role Analysis, Synthesis and Structuring) [WiJ90] is an object-oriented design methodology consisting of three operations. These operations are based on the encapsulation, inheritance and dynamic binding properties of object-orientation. Analysis describes sub-problems by encapsulating behaviour in the objects of an object-model, which is termed a Role model. Synthesis defines composite objects by inheriting behaviour from several simpler objects. Structure specification prescribes how objects can be bound together in an actual instance of a system.

Lieberherr et al. [LHR88], [LiH89] have developed an object-oriented CASE tool, called the Demeter system, which generates C++ [ElS90] or Flavors [Moo86 ] class definitions from language-independent class dictionaries. Class dictionaries describe the part-of and inheritance relationships between classes. The Demeter system includes tools for checking design rules and for implementing design.

Rumbaugh et al. [RBP+91] have developed a design methodology called Object Modelling Technique (OMT) which covers the entire development life-cycle: analysis, design, and implementation. The methodology has the following steps: write or obtain an initial description of the problem, build an object model, develop a dynamic model, construct a functional model, combine the three models to obtain the operations on classes, design algorithms to implement the operations.

# CHAPTER 3

# THE STATIC MODEL: SEMANTICS

## 3.1 Extending the Entity Relationship Model

The entity relationship model was first proposed by Chen [Che76]. A number of researchers have proposed extensions to the model. In this section we present another extension to the entity relationship model. This extension is part of the static model for our design methodology which is presented in Chapter 4.

### 3.1.1 Relationships between classes

A relationship is an association between the objects in one or more classes. The degree of a relationship is the number of classes participating in the relationship. A unary relationship is an association between the objects of a single class, a binary relationship is an association between the objects of two classes, and a ternary relationship is an association between the objects of three classes. Relationships of degree greater than three are not common. Two main constraints can be placed on relationships : cardinality ratio constraint and participation constraint. The participation of the entities in a relationship can be total or partial. A total participation of a class in a relationship means that every instance of that class must participate in the relationship. A partial participation means that some instances of the class may not participate in the relationship. The cardinality ratio constraint specifies how many instances of one class may relate to a single instance of an associated class.

A relationship between two classes is denoted by a diamond shaped box with lines connecting the two classes. The name of the relationship is placed inside this box. We denote total participation of a class in a relation by a shaded circle at the end of an association line; partial participation is denoted by a circle. The cardinality of a class is usually written at the end of an association line. We use the notation shown below for cardinality. In the notations k, m, and n are integers such that $k \geq 1$, $n, m \geq 0$ and $m < n$.

| **Notation** | **Meaning** |
|---|---|
| $n$ | exactly n instances |
| $n^+$ | n or more instances |
| $n^-$ | n or less instances |
| $n_k^+$ | n or n+2k or n+3k or ... instances. |
| $n_k^-$ | n or n-2k or n-3k or ... instances. |
| m_n | m, m+1, m+2, ... or n instances. |
| m, n, k | m or n or k instances. |
| m/n | m instances participate in a relationship but at any instant exactly n are executing concurrently |
| $m/n^-$ | m instances participate in a relationship but at any instant a maximum of n can execute concurrently |
| $m/n^-[c]$ | m instances participate in a relationship but at any instant a maximum of n can execute concurrently provided condition c is true. |

Note: The only exception to the notation $0^+$, denoting zero or more instances, is when we indicate a many to many relationship. In that case instead of using $0^+ : 0^+$ we use $(0,1)^+ : (0,1)^+$. The reason for choosing this notation will be apparent below when the notion of exceptions on cardinalities is introduced.

### 3.1.2 Exceptions on cardinalities

Sometimes we may wish to indicate that for given possible cardinality values, some values occur only as exceptions, i.e. few class instances participate in a given relationship with such cardinalities. The following notations are used, where k, m, n and p are integers such that $k \geq 1$, $m, n, p \geq 0$, and $m < n < p$:

| Notation | Meaning |
|---|---|
| $n_k^{+e}$ | n is the normal cardinality, n+k, n+2k, ... are exceptional. |
| $n_{ke}^{+}$ | n+k, n+2k, ... are the normal cardinalities, n is exceptional. |
| $n_k^{-e}$ | n is the normal cardinality, n-k, n-2k, ... are exceptional. |
| $n_{ke}^{-}$ | n-k, n-2k, ... are the normal cardinalities, n is exceptional. |
| $n_k^{+-e}$ | n is the normal cardinality, n-k, n-2k, ... and n+k, n+2k, ... are exceptional. |
| $n_{ke}^{+-}$ | n-k, n-2k, ... and n+k, n+2k, ... are the normal cardinalities, n is exceptional. |
| $k^{+e}$ | k is the normal cardinality, k+1, k+2, ... are exceptional. |
| $k_e^{+}$ | k+1, k+2, ... are the normal cardinalities, k is exceptional. |

|  |  |
|---|---|
| $k^{+-e}$ | k is the normal cardinality, k-1, k-2, ... and k+1, k+2,... exceptional. |
| $k_e^{+-}$ | k-1, k-2, ... and k+1, k+2, ... are the normal cardinalities, k is exceptional. |
| $m, n_e, k$ | m and k are the normal cardinalities, n is exceptional. |
| $m\_p, n_e$ | every integer in the range m to p is a normal cardinality except p. |

As an example consider a relationship between a class A and a class B with cardinality ratio $1^+ : 1^+$. We can define eight different cardinality ratios : $1^{+e} : 1^{+e}$, $1e^+ : 1e^+$, $1^{+e} : 1e^+$, $1e^+ : 1^{+e}$, $1^{+e} : 1^+$, $1^+ : 1^{+e}$, $1e^+ : 1^+$, $1^+ : 1e^+$.

For the many to many cardinality ratio $(0,1)^+ : (0,1)^+$ we can define exceptions on either an entire bracket or on individual digits. For example :

$(0,1)e^+ : (0,1)e^+$ means that $2^+ : 2^+$ is the normal cardinality ratio.

$(0,1)^{+e} : (0,1)e^+$ means that $1^- : 2^+$ is the normal cardinality ratio.

$(0e,1)^+ : (0e,1)^+$ means that $1^+ : 1^+$ is the normal cardinality ratio.

$(0e,1)^+ : (0,1)e^+$ means that $1^+ : 2^+$ is the normal cardinality ratio.

### 3.1.3 Cardinality ratios for ternary relationships

Unless ternary relationships are given stringent semantic interpretation they can be ambiguous. To remove any ambiguity on the semantics of a ternary relationship we use nine cardinality ratios. Three of these cardinality ratios relate the three pairs of classes. Each of the remaining six ratios relate one instance of a particular class to instances of another class taking into account the third class. We use the following notation for each of these six other ratios : 1 INSTANCE OF CLASS-1 : N INSTANCES OF CLASS-2 (1INSTANCE OF CLASS-2 : K INSTANCES OF CLASS-3). The semantics we attach to this notation is that one instance of CLASS-1 is related to N instances of CLASS-2, and each instance of CLASS-2 related to one instance of CLASS-1 is related to K instances of CLASS-3. As an example consider the ternary relationship shown in Figure 3.1. The three cardinality ratios shown in Figure 3.1 are interpreted as follows:

One teacher can teach zero or more courses.

One course can be taught by zero or more teachers.

One teacher can teach zero or more students.

One student can be taught by zero or more teachers.

One course can have zero or more students.

One student can take zero or more courses.

These six interpretations are not sufficient to explain the semantics of the ternary relation without ambiguity. For example, we know that one teacher can teach many courses and that a student can take many courses; but how do we specify the restriction that a student cannot be taught a particular course by more than one teacher ? To answer questions such as this we define the remaining six cardinality relations. In each case we give two different interpretations. In the following cardinality relations, we use T to denote Teacher, C to denote Course, and S to denote Student:

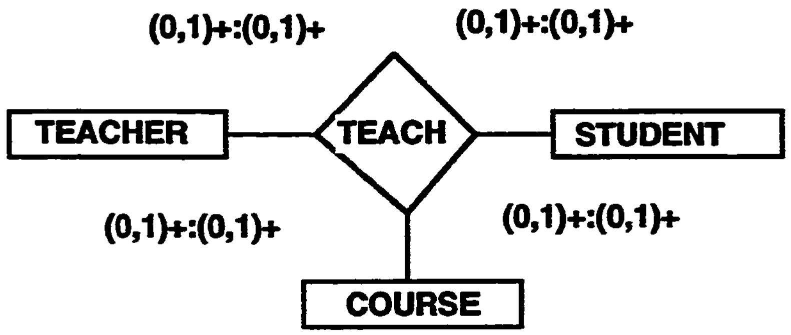


**Figure 3.1: Ternary relationship**

TEACHER-STUDENT PAIR:

**Semantics**

$1_T{:}0_S{}^+(1_S{:}1_C{}^+)$

One teacher can teach zero or more students.

One student can be taught by one particular teacher one or more courses.

$1_S{:}0_T{}^+(1_T{:}1_C{}^+)$

One student can be taught by zero or more teachers.

One teacher can teach a particular student one or more courses.

**Alternative semantics**

$1_T{:}0_S{}^+(1_S{:}1_C)$

One teacher can teach zero or more students.

One student can be taught by a teacher one course only.

$1_S{:}0_T{}^+(1_T{:}1_C)$

One student can be taught by zero or more teachers .

One teacher can teach a particular student one course only.

TEACHER-COURSE PAIR:

$1_T{:}0_C{}^+(1_C{:}1_S{}^+)$ $1_T{:}0_C{}^+(1_C{:}1_S)$

A teacher can teach zero or more courses.

A course taught by one teacher can have one or more students.

A teacher can teach zero or more courses.

A course taught by one teacher can have one student only.

$1_C{:}0_T{}^+(1_T{:}1_S{}^+)$

A course can be taught by zero or more teachers.

A teacher can teach one course to one or more students.

$1_C{:}0_T{}^+(1_T{:}1_S)$

A course can be taught by zero or more teachers.

A teacher can teach a course to one student only.

STUDENT-COURSE PAIR:

$1_S{:}0_C{}^{+}(1_C{:}1_T{}^{+})$

A student can take zero or more courses.

A student can be taught a particular course by one or more teachers.

$1_S{:}0_C{}^{+}(1_C{:}1_T)$

A student can take zero or more courses.

A student can be taught a particular course by one teacher only.

$1_C{:}0_S{}^{+}(1_S{:}1_T{}^{+})$

A course can be taken by zero or more students.

A student can be taught a particular course by one or more teachers.

$1_C{:}0_S{}^{+}(1_S{:}1_T)$

A course can be taken by zero or more students.

A student can be taught a particular course by one teacher only.

### 3.1.4 Dependent relationships

We use the notation in Figure 3.2 to denote that relation R1 and R2 must occur together, and the notation in Figure 3.3 to denote that they cannot occur together. For example, Figure 3.4 has the semantics: Every person studying in a primary school is not married, and every married person is not studying in a primary school.

### 3.1.5 OR-association links

We use the notation in Figure 3.5 to denote that either class C1 or class C2 is related to class C3 through relation R.

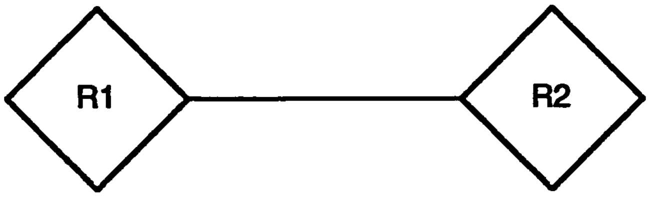


Figure 3.2: Inclusive relations

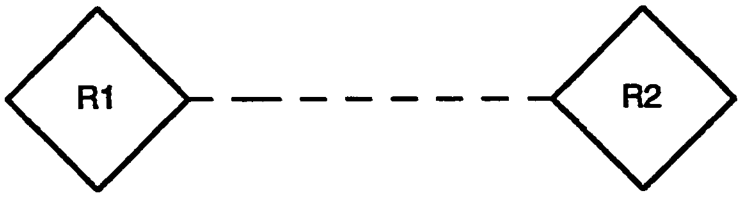


Figure 3.3: Exclusive relations

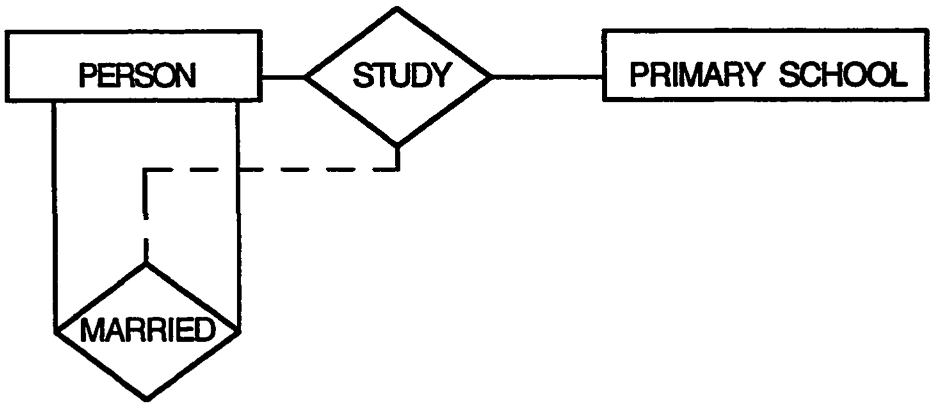


Figure 3.4: Exclusive relations: An example

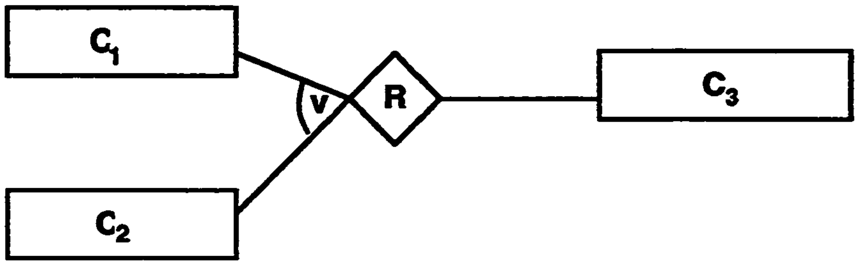


**Figure 3.5: An OR-association**

## 3.2 The is-a relationship: Notation and semantics

In Chapter 2 we defined the is-a (inheritance) relationship between two or more classes. We also gave definitions for a concrete class, an abstract class, a deferred abstract class, and a partially deferred abstract class. The notations for these different types of classes are given in Figure 3.6.

The is-a relationship between a superclass and a subclass is represented by a shaded arrow pointing from the subclass to the superclass as in Figure 3.7. An alternative notation for an is-a relationship between a superclass and a number of subclasses is given in Figure 3.8. The meaning of the symbols v and $\wedge$ in Figure 3.7 and 3.8 is given below.

The is-a relationship is transitive, anti-symmetric, and trivially reflexive. Extending on the work of Abiteboul and Hall [AbH87] and that of Elmasri and Navathe [ElN89] on is-a semantics we define twelve types of is-a relationships between a superclass and its subclasses:

1. Complete, total, exclusive
2. Incomplete, total, exclusive
3. Complete, partial, exclusive
4. Incomplete, partial, exclusive
5. Complete, total, inclusive
6. Incomplete, total, inclusive
7. Complete, partial, inclusive
8. Incomplete, partial, inclusive
9. Complete, total, exclusive-inclusive

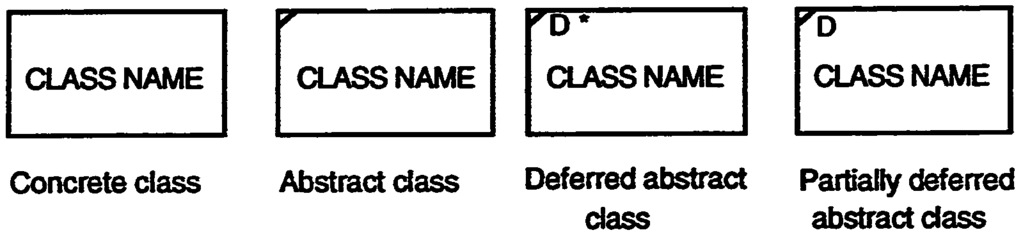


Figure 3.6: Class notation

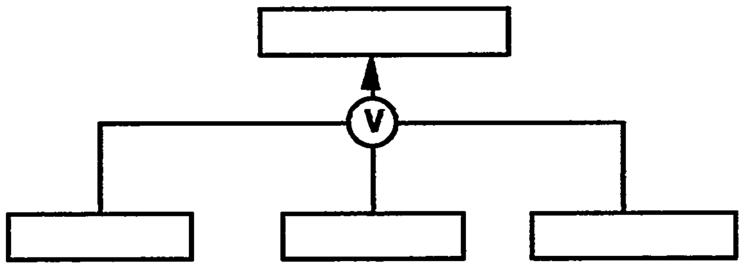


Figure 3.7: Exclusive is-a

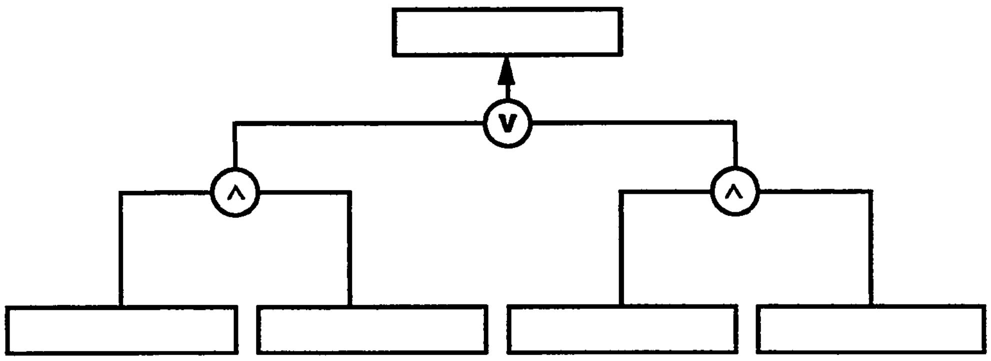


Figure 3.8: Exclusive-inclusive is-a

10. Incomplete, total,exclusive- inclusive
11. Complete, partial, exclusive-inclusive
12. Incomplete, partial, exclusive-inclusive

A superclass has a complete is-a relationship with its direct subclasses if these subclasses are the only specialisations of interest for the class. We may have several specialisations of the same class based on different distinguishing characteristics of the instances of the class. By introducing the notion of complete is-a relationship we are putting restriction on the way we can specialise a class. This is one way of controlling careless subclassing as well as controlling unwarranted modifications to the application being designed. A superclass has an incomplete is-a relationship with its direct subclasses if other specialisations for the class may be allowed in future modifications.

A class has a total is-a relationship with respect to a given specialisation if every instance of the class is an instance of some direct subclass in the specialisation. In a partial is-a relationship an instance of a class may not belong to any of the direct subclasses in the specialisation.

A class has an exclusive is-a relationship with its direct subclasses if any two or more of these subclasses cannot have a common subclass. It has an inclusive is-a relationship with its direct subclasses if any two or more of the subclasses can have a common subclass. A class has an exclusive-inclusive is-a relationship with its direct subclasses if some of these direct subclasses can have common subclasses while others cannot.

We use a shaded circle at the head of the is-a relationship arrow to denote total is-a relationship, a circle to denote partial is-a relationship, the letter c to denote complete is-a relationship , and the symbols $\neg c$ to denote incomplete is-a relationship. We denote exclusive is-a relationship by using the symbol V in the is-a relationship arrow as in Figure 3.7, and we denote inclusive is-a relationship by the symbol $\wedge$ as in Figure 3.8. A typical notation for exclusive-inclusive relationships is shown in Figure 3.8 .

## 3.3 The is-part-of and has-part relationships

A class may be predefined without any attributes, or it may have a number of attributes, and any of the attributes may have as their domains other non-predefined classes. A composite class is a class with one or more attributes with domains which are related to the class by the is-part of relationship.

### 3.3.1 Is-part-of semantics

The is-part-of relationship is transitive, antisymmetric, and non reflexive. Extending on the work of Kim et al. [KBG89] on composite objects we define six types of is-part-of relationships:

1. Dependent exclusive:

   A class B is a dependent exclusive component of a class A if B is a component of A only, and if the existence of B depends on the existence of A.

2. Independent exclusive:

   A class B is an independent exclusive component of a class A if B is a component of A only, and if the existence of B does not depend on the existence of A.

3. Single-dependent shared:

   A class B is a single-dependent shared component of a class A if B is a component of A and possibly other classes, and if the existence of B depends on the existence of A only.

4. AND-dependent shared:

   A class B is an AND-dependent shared component of a class A if B is a component of A and one or more other classes, and if the existence of B depends on the existence of A and at least one other class of which it is a component.

5. OR-dependent shared:

   A class B is an OR-dependent shared component of a class A if B is a component of A and possibly other classes, and if the existence of B depends on either the existence of A or on the existence of at least one other class of which it is a component.

6. Independent shared:

   A class B is an independent shared component of a class A if B is a component of A and possibly other classes, and if the existence of B does not depend on the existence of A .

The is-part-of relationship between a component class and its composite class is represented by an arrow, bearing a head with a letter P, pointing from the component class to the composite class. We use a shaded circle at the tail of this arrow to denote dependent is-part-of relationship, a circle to denote independent is-part-of relationship, the letter s to denote shared is-part-of relationship , the symbols $\neg$s to denote exclusive is-part-of relationship , the symbols vs to denote OR-shared is-part-of relationship, and the symbols $\wedge$s to denote AND-shared is-part-of relationship.

The cardinality relationship between a component and its composite class specify how many instances of the component class make up one instance of the composite class.

As an example on the above notations, Figure 3.9 shows a dependent exclusive is-part-of relationship and an AND-dependent shared is-part-of relationship.

## 3.3.2 Has-part semantics

There are five types of has-part relationships:

1. Dependent exclusive:

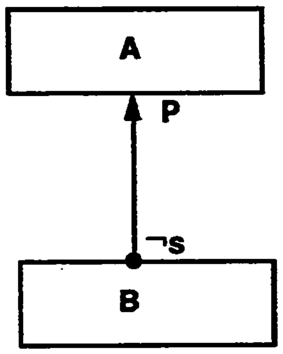
   A class A is a dependent exclusive composite of a class B if B is a component of A only, and if the existence of A depends on the existence of B.

2. Independent exclusive:

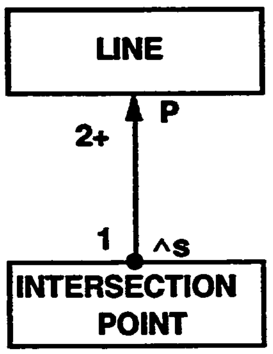
   A class A is an independent exclusive composite of a class B if B is a component of A only, and if the existence of A does not depend on the existence of B.

3. Single-dependent shared:

   A class A is a single-dependent shared composite of a class B if B is a component of A and possibly other classes, and if the existence of A depends on the existence of B.



Dependent exclusive



AND-dependent shared

Figure 3.9: Part-of relationships

4. AND-dependent shared:
   A class A is an AND-dependent shared composite of a class B if B is a component of A and one or more other classes, and if the existence of A and at least one other composite class of B depends on the existence of B.
5. Independent shared:
   A class A is an independent shared composite of a class B if B is a component of A and possibly one or more other classes, and if the existence of A does not depend on the existence of B.

The notations used for has-part relationships are the same as those for is-part-of relationships except that the symbols are placed at the tips of the arrows representing the is-part-of relationships.

### 3.3.3 Constraints on composite class topologies

1. A component class can participate in at most one independent exclusive is-part-of relationship.
2. A component class can participate in at most one dependent exclusive is-part-of relationship.
3. If a class is a component in an independent exclusive is-part-of relationship, then it cannot be a component in an exclusive dependent is-part-of relationship with another class; and vice versa.
4. If a class is a component in a dependent or independent exclusive is-part-of relationship, then it cannot be a component in a dependent or independent shared is-part-of relationship with other class; and vice versa.

### 3.3.4 Make-component rule

A class B can be made a component of a class A through an attribute T of A if the following conditions are satisfied:

1. If T is an exclusive composite attribute, B must not be a component in any other exclusive or shared is-part-of relationship.
2. If T is a shared composite attribute, B must not already be a component in any other exclusive is-part-of relationship.

### 3.3.5 Semantics of deletion for composite classes

If a class B is a component of a class A, then the deletion of A implies the deletion of B if any of the following conditions holds:

1. B is a dependent exclusive component of A.
2. B is a single-dependent shared component of A.
3. B is an AND-dependent shared component of A.
4. B is an OR-dependent shared component of A and there is no other composite class to which B is an OR-dependent shared component.

If a class B is a component of a class A, then the deletion of B implies the deletion of A if any of the following conditions holds:

1. A is a dependent exclusive composite class of B.
2. A is a single-dependent shared composite class of B.
3. A is an AND-dependent shared composite class of B.

### 3.3.6 Recursive composite classes

A composite class is recursive if it consists of both exclusive is-a and is-part-of relationships such that it has one is-a specialisation in which one subclass is a composite class having as its component one of its superclasses. We use the notation in Figure 3.10 to denote a recursive composite class.

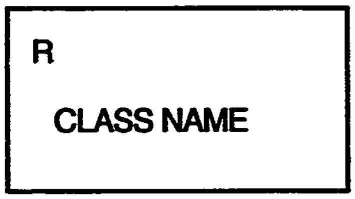


**Figure 3.10 : A recursive composite class.**

# CHAPTER 4
# THE DESIGN METHODOLOGY

## 4.1 Outline of the design methodology

Our design methodology consists of the following seven stages:

1. Exploratory stage.
2. Division of the design task into subtasks.
3. Extraction of additional subsystems and classes.
4. Construction of class interfaces.
5. Evaluation and refinement of classes and subsystems.
6. Construction of the dynamic-model.
7. Integration of the design subtasks.

## 4.2 The exploratory stage

The input to this stage is the requirements specification document if it is available, otherwise a complete object-oriented analysis is carried out. A group workshop approach is used in conducting this exploratory stage. The design team is divided into a number of groups. Each group considers all the requirements and tries to come up with an initial list of subsystems, classes, and their collaborations without performing a detailed analysis of the requirements. Members are then chosen from these groups to merge the different lists into one list. Based on this tentative list of subsystems, classes, and their collaborations a criteria for the subdivision of the design task is reached. The purpose of this stage is to give all the designers an overview of the problem domain, to supply a set of design tasks for division amongst the design team, and to supply a better understanding of the complexity of each design task and the resources necessary for that task.

This stage has seven substages:

1.Determination of system behaviours.
2. Construction of event-response lists.
3. Determination of the external interfaces to the system.
4. Determination of an initial list of subsystems.
5. Determination of an initial list of classes and class lattices/hierarchies.
6. Construction of an initial list of CRC (Class, Responsibility, Collaboration) cards, subsystem specification cards, class specification sheets, entity relationship diagrams, and inheritance diagrams.
7. Integration of the class and subsystem lists found by different design teams.

### 4.2.1 Determination of system behaviours

1. From the requirements specification elicit a list of desired behaviours of the system. These behaviours define the system roles and responsibilities. Find answers to the following questions:
   - What are the main roles of the system ?
   - What are the main responsibilities under each role ?
2. Find system behaviours from the expected usage of the system,i.e.,
   - What do users expect the behaviour of the system to be ?
3. Find system behaviours from domain knowledge. This includes domain experts, widely used literature on the subject, and previous experience with the type of problem being solved.

4. Find major behaviours of known similar systems,i.e.,
   - Does the system to be designed exclude any behaviour of a similar system ? If yes, why ? Is there a need to modify the behaviour list of the system to include this behaviour ?
   - Does the system to be designed have any behaviour not found in all or most known similar systems ? If yes, why ? Is there a need to modify the behaviour list of the system to exclude this behaviour ?
   - Is there a need at all to design a new system ? If yes, why ? How different is the proposed system from current systems ?

### 4.2.2 Construction of event-response lists

From the list of behaviours found above determine the external behaviours of the system. From this list construct event-response lists. An event-response list is a description of possible requests to the system along with accompanying system reactions.

### 4.2.3 Determination of the external Interfaces to the system

1. Find the classes required to model user interfaces.
2. Find the classes required to model the interfaces to external hardware devices.
3. Find the classes required to model the interfaces to external databases if any.
4. Find the collaborator classes of the interface classes if the process of doing so will not require detailed analysis. Collaborations represent requests from a client class to a server class in fulfilment of a client responsibility. For the

most part classes that represent external interfaces do not require the services of other classes in the system; they are server classes.

### 4.2.4 Determination of an initial list of subsystems

A subsystem is a group of classes that collaborate among themselves to provide a clearly delimited unit of functionality. A subsystem may contain other subsystems. The division of an application into a number of subsystems is crucial in managing the complexity of the design process. Answers to questions like those given below are helpful in determining subsystems:

- Are any subsystems apparent or implied from the requirements ?
- Is there any system behaviour which will be better realized as a subsystem ?
- Do any of the interface classes form a subsystem ?

### 4.2.5 Determination of an initial list of classes and class lattices/hierarchies

1. From domain knowledge and past experience on similar problems try to find off- the-shelf classes.
   - What modifications, if any, do these classes require or what of their features need to be redefined in their subclasses for them to fit into the application being designed ?
2. For each of the major system responsibilities found in 4.2.1 try to find the classes satisfying each responsibility.
   - What are the categories of these classes ?,i.e., do these classes form any abstractions ?

- What does each of these classes collaborate with in order to accomplish each of its responsibilities ?
- What are the categories of these collaborator classes ?

3. For each class found determine if it belongs to any of the identified subsystems.
4. Determine any is-a, has-parts and is-part-of semantics of the classes found.
5. Try to determine the types of each class found. Apart from being either concrete, or abstract, or deferred abstract, or partially deferred abstract, a class can be composite, recursive composite, actor, agent, server, or helper. Actor classes are classes whose instances send messages to instances of other classes based on an internal or external stimulus. Agent classes are classes whose instances provide services and may send messages to other instances. Instances of an agent class are passive in the sense that they do not initiate action without being called by instances of an actor class or another agent class. Server classes are classes whose instances provide services but do not send messages to other instances. A helper class is a class whose instances help the instances of another class to provide its services.
6. Try to find the attributes of each class found.

### 4.2.6 Construction of an initial list of CRC cards, subsystem specification cards, class specification sheets, entity relationship diagrams, and inheritance diagrams.

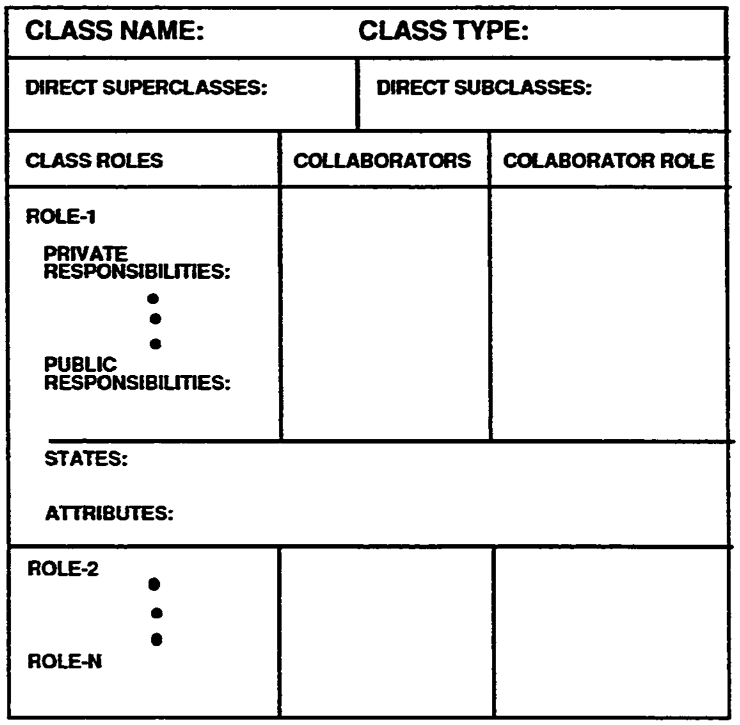
1. Start to draw inheritance diagrams for classes related by the is-a relationship. Indicate the semantics of the is-a relationship for each class

having more than one subclass whenever appropriate. Check that each inheritance lattice/hierarchy is structurally consistent (We develop the design consistency checking algorithms to be applied in Chapter 5). Use inheritance diagrams to capture the inheritance structure only. Do not fill class diagrams with method and attribute names. Record them on CRC cards and class specification sheets instead.

2. Start to draw composite classes, if any, and indicate their has-parts and is-part-of semantics. Make sure that there are no is-part-of semantic conflicts.
3. Start to draw the entity relationship diagrams for any related classes found. The approach is not to draw one, huge entity relationship diagram; but a number of diagrams in which classes bound by closely related relationships appear in a single diagram. Although this approach will produce many separate entity relationship diagrams, it allows designers to deal with groups of closely related classes one at a time. Do not indicate class attributes in these diagrams. As mentioned, attributes are recorded on CRC cards and class specification sheets. The sole purpose of using entity relationship diagrams is to capture as much of the semantics of the application as possible. Most semantics of interest, possible exceptions, constraints, and actions to be taken in case of exceptions or constraint violations for a group of related classes are recorded on the corresponding entity relationship diagram.
4. Start to construct a CRC card for each class found. We use a modified form of CRC cards to that presented in both [BeC89] and [WWW90]. In our cards we group the responsibilities of a class under class roles, we indicate the class attributes in each role, we indicate role states, and we include the roles of

the collaborator classes. The intent is not to try to find all these parameters in one design stage, but rather to do so in stages as the design process evolves. The format of our CRC card is shown in Figure 4.1. On the back of a CRC card we write a short description of the overall purpose of the class. Note that we do not record on a CRC card inherited roles and that a class can have a subsystem as a collaborator if it is collaborating with a class inside the subsystem. The responsibilities under a role can be private or public. A private responsibility represents a class behaviour which cannot be requested by other classes, whereas a public responsibility represents a class behaviour which can be requested by other classes.

5. Start to construct a class specification sheet for each class found. At this stage only some fields of a class specification sheet can be filled. The fields of a class specification sheet are shown in Figure 4.2. Later on in stage four of the design process, after the construction of class interfaces, each non-inherited class role will be expanded as in Figure 4.3.
6. Start to construct a subsystem specification card for each subsystem found. A subsystem supports public responsibilities just as a class. The public responsibilities of a subsystem are the public responsibilities of its classes or its subsystems that provide services to clients outside the subsystem. Record any subsystem public responsibility on the subsystem card. Besides each public responsibility, record the internal class or subsystem that actually supports the responsibility. The format of a subsystem specification card is shown in Figure 4.4.



| | | |
|---|---|---|
| CLASS NAME: | CLASS TYPE: | |
| DIRECT SUPERCLASSES: | DIRECT SUBCLASSES: | |
| CLASS ROLES | COLLABORATORS | COLABORATOR ROLE |
| ROLE-1<br>PRIVATE RESPONSIBILITIES:<br>•<br>•<br>•<br>PUBLIC RESPONSIBILITIES: | | |
| STATES:<br>ATTRIBUTES: | | |
| ROLE-2<br>•<br>•<br>•<br>ROLE-N | | |

Figure 4.1: A CRC card

CLASS NAME :

CLASS TYPE :

VERSION NUMBER :

CLASS DESCRIPTION :

INHERITANCE DIAGRAM : PAGE #

STATECHART : PAGE #

DIRECT SUPERCLASSES :

DIRECT SUBCLASSES :

SUBCLASS IS-A SEMANTICS :

ATTRIBUTES :

STATES :

CLASS IS PART OF :

IS-PART-OF SEMANTICS :

CLASS HAS PARTS :

HAS-PARTS SEMANTICS :

INHERITED ROLES :

NON-INHERITED ROLES :

COMMENTS:

**Figure 4.2 : A class specification sheet.**

ROLE NAME:
DESCRIPTION :
PRIVATE RESPONSIBILITIES :
PRIVATE RESPONSIBILITY-1
    Signature-1:
        Description:
        List of collaborations:
    Signature-2:
        Description:
        List of collaborations:
    Signature-3:
        •
        •
        •
    Signature-n:
PRIVATE RESPONSIBILITY-2:
        •
        •
        •
PRIVATE RESPONSIBILITY-M:
        •
        •
        •
PUBLIC RESPONSIBILITIES:
PUBLIC RESPONSIBILITY-1:
        •
        •
        •
PUBLIC RESPONSIBILITY-K:

**Figure 4.3: Expanded Role field**

| | |
|---|---|
| SUBSYSTEM NAME: | |
| DESCRIPTION: | |
| ENCAPSULATED SUBSYSTEMS: | |
| DIRECTLY ENCAPSULATED CLASSES: | |
| ROLES: | |
| PUBLIC RESPONSIBILITY | SUPPORTING CLASS OR SUBSYSTEM |
| | |
| | |
| ● ● ● | |

Figure 4.4: Subsystem specification card

### 4.2.7 Integration of the class and subsystem lists found by different design teams.

In this preliminary stage the best class-design and subsystem-design alternatives are chosen. If two or more class designs are deemed equally good they are assigned different version numbers. New classes are designed if necessary by generalising, specialising, composing or decomposing some of the classes in the lists. Class names are chosen carefully and each class is given a unique name. The proper selection of class names early in the design process greatly simplifies and facilitates later design steps. New subsystems may be generated in this stage. Necessary additions are made to the CRC cards and the specification sheets of the chosen classes. Similarly necessary changes are made to the subsystem specification card of each chosen subsystem if it undergoes any modifications.

## 4.3 The division of the design task into subtasks

From the class and subsystem list formed above, the representative team of designers:

1. Determines what system requirements the subsystems obtained so far fulfil.
2. Divides the remaining system requirements into small related subsets, paying particular attention to the functional and operational requirements of the system.
3. Determines which subset contains classes which are not members of any subsystem found so far.

4. Determines any classes appearing in more than one subset. These classes will be assigned to more than one design team to be developed independently. Later on they will be integrated.
5. Figures out the complexity of each subsystem and each requirements subset and the resources required to design them. This is a difficult process heavily dependent on the past experience of the design team.
6. Divides the design task into design subtasks accordingly.
7. Assigns the design subtasks to different groups of designers. Each design subtask will at this point contain a number of identified classes and possibly a number of identified subsystems.

## 4.4 Extraction of additional subsystems and classes from each design subtask

Our approach in this stage is to use some of the procedures used in the preliminary stage, but a more detailed analysis than that carried out in that stage is carried out here. Also, for each design team, the analysis is only directed to the assigned design subtask. The emphasis is on getting answers to the questions: What are the roles of each class ? What are the responsibilities for each role ? Can a class perform any of its responsibilities independently without collaborating with other classes ? What are the collaborators of each class ? For a given responsibility of a class and the corresponding collaborator what is the role of the collaborator ? What are the attributes required in each role ? What are the role-states ?

Since the requirements subsets are much smaller than the entire requirements specification, we may augment the process of finding classes and subsystems by Booch's lexical method for identifying problem-domain entities and their operations [Boo83]. The procedure is to create a list of the key nouns and noun phrases, a list of key verbs, verb phrases, and adjectival phrases, and a list of adjectives, adjectival phrases, adverbs, and adverbial phrases. The first list serves as the first approximation to the problem-domain entities and some of their attributes, the second to their operations, and the third, to most of their attributes.

It is likely that of the classes found so far, some will be abstract and others concrete. From these classes we can obtain more classes by the inheritance relationship. For a class to be related by inheritance to another class, there should be some relationship of functionality and data between the two classes.We identify abstract superclasses by grouping concrete classes having common behaviours and common attributes. We identify more abstract classes by grouping related abstract classes into more abstract superclasses. We identify concrete subclasses from abstract classes by progressively specialising these classes into less and less abstract classes.

As we go on deriving abstract classes from concrete classes, and concrete classes from abstract ones, we specify the is-a semantics between pairs of direct subclasses of each class if appropriate. The design process will necessitate making changes to the class lattices by adding, deleting, or redesigning classes. We must make sure that the structural consistency of the class lattices are not violated (use the design consistency checking algorithms given in chapter 5).

It is important that we assign each of the smaller design tasks to more than one designer, so that we can obtain design alternatives. We then evaluate these design alternatives and choose the best one, at the same time we record why any meaningful alternative was rejected.

As we do the above process of abstraction and specialisation we follow the following class design guidelines :

1. Select meaningful class names.

   Each class name must be unique. Choose names that are descriptive of the responsibilities attached to the classes. Avoid abbreviations that may confuse other designers.
2. Do not overuse inheritance.

   Overuse of inheritance results in an application which is hard to understand by having too many classes with duplicate functionality. Two courses of action are possible if two or more classes have duplicate functionality, either merge the classes into one new class, or make the classes subclasses of a common superclass providing the shared functionality.
3. Do not use inheritance inappropriately.

   Do not use inheritance between classes when the relationship between them is not is-a; but is either is-part-of relationship or another relationship.

4. Do not create abstract classes inheriting from concrete classes.

   Concrete classes can be instantiated whereas abstract classes cannot. Therefore an abstract class should never inherit from an abstract class.

5. Do not create classes with excessive coupling unless necessary.

   A class should be dependent on as few other classes as possible. Fewer collaborations means that the class is less likely to be affected by changes to other parts of the system.

6. Do not create classes with excessive responsibilities.

   If a class has too many responsibilities perhaps it can be split into two or more classes. Often a portion of class behaviour can be abstracted out and assigned to one or more helper classes. A good design balances the goal of having a small number of classes with the conflicting goal of having a small number of classes whose relationship with other classes can be easily grasped.

7. Do not create classes with unconnected responsibilities.

   The responsibilities of a class must be connected by data, functionality or any other obvious binding. A class representing the behaviour of two or more concepts should be separated into two or more distinct classes.

8. Do not create classes with no responsibilities unless they are deferred classes.

   If a class inherits a responsibility that it will implement in a unique way, then it adds functionality despite having no responsibilities of its own. On the other hand, abstract classes that define no responsibilities and which will not be implemented as deferred classes should be discarded.

9.The only members of the public interface of a class should be the public methods of a class.

From the classes identified and their collaborations, we try to identify subsystems. Subsystems are identified by finding a group of classes, each of which fulfils different responsibilities; but which collaborate closely to fulfil a greater responsibility. A class is part of a subsystem only if it exists solely to fulfil the responsibilities of that subsystem.

At the end of this design stage, we should have a set of classes that match the objects in the problem space. Their attributes and responsibilities are well defined and their collaborations have been identified. Together with this we should have identified most of the subsystems in our application. We update the class inheritance diagrams, entity-relationship diagrams, CRC cards, class specification sheets, and subsystem specification cards.

## 4.5 The construction of class interfaces.

Consider one role of a class at a time. Turn each role responsibility into a set of signatures. The signature of a method defines the interface to the method: the number of arguments it requires, their order and their types and the number, order and types of the values it returns. Each responsibility will have one or several messages associated with it. Use different scenarios, including exceptional conditions, to generate the list of messages associated with each responsibility. For each message sent by the class use the information recorded on the CRC card for the class to determine the receiving class. Turn the corresponding responsibility in this receiving class into a signature. Again consider different scenarios of the requests this receiving class can demand on the sender class in order to generate more messages. Similarly for each message received by the class, consider the sending class. Turn the corresponding responsibility into a set of signatures by considering different scenarios. In this way we ensure that each message will have a sender and a receiver. Name the methods and messages carefully. Make sure the names are consistent and accurately describe the methods and messages they label. Use short names if possible; but avoid abbreviations that may confuse other designers. Record the signatures and messages for each class on the class specification sheet for the class. Write short descriptions for the signatures and a list of their collaborations. Repeat this process for each class in the design subtask.

## 4.6 Evaluation and refinement of classes and subsystems

In this stage we check that the class-design and subsystem-design guidelines have been adhered to. The responsibilities and collaborations of classes and subsystems should be validated against the original requirements. Classes are tested by scenario, evaluated and then revisions made if required. The process of class and subsystem evaluation is iterative. All throughout the process classes may be added, deleted, composed, decomposed, or altered. We must make sure that throughout the process the structural consistency of class lattices is preserved.

## 4.7 The construction of the dynamic-model.

We use Harel's Statechart notation [Har88] to construct the dynamic model of our application. Statecharts are an extension of state-transition diagrams. They address the problem of the combinatorial explosion of transitions which can result when state-transition diagrams are used to represent complex systems.

### 4.7.1 Harel's Statechart notation

Statecharts introduce hierarchy and concurrency into state-transition diagrams. Hierarchy allows states with common transitions to be abstracted into an exclusive-OR state. An object in an exclusive-OR state can only be in one substate of the state. Concurrency can be represented by an AND-state. A simple AND-state consists of two or more concurrent exclusive-OR states. An object in a simple AND-state must be concurrently executing all its exclusive-OR components. Thus an AND-state represents the Cartesian product of all its substates. Concurrent states generally correspond to composite objects with interacting parts. Arbitrarily complex state structures can be constructed out of combinations of exclusive-OR and AND-states.

A state is represented by a closed contour in the form of a rounded box. A superstate encloses all its substates. The contour of an AND-state is partitioned into two or more segments by dotted lines; each segment represents a substate. A superstate can have a default initial state. It is represented by a small arrow, with no source state, ending at the contour of the default initial state. A transition originating at the boundary of a superstate applies to all substates in the state. A transition ending at the boundary of a superstate applies to the default initial state only unless the transition is multiple conditional in which case it is applicable to all substates, with each transition being fired if its guard is true. A transition out of an AND-superstate from any of its substates has the effect of terminating execution in all other substates. Transitions in the components of an AND-state may depend on each other or they may be independent.

In the following state-transition diagrams and statecharts we use non-bracketed lower case letters for event-messages that trigger the transitions. bracketed lower case letters are conditions on guarded transitions. A bracketed condition in the form [in(S)] means that the particular guarded transition of a substate can only fire when the object is in state S in another substate. The state-transition diagram in Figure 4.5 can be abstracted into the exclusive-OR state X shown in Figure 4.6. The AND-state Y and states H and I in Figure 4.7 represent the state-transition diagram in Figure 4.8. Figure 4.7 and 4.8 are modified from [Har88].

### 4.7.2 Extension to Harel's notation

We represent dependencies between transitions and states in other substates by joining the transition to the state. We use the notion of conduits in contours for AND-states. Instead of subdividing the contour directly by dotted lines as in Figure 4.9 we partition it as in Figure 4.10 in which conduits are left between the segments. We use these conduits to direct the lines joining dependent transition-arc and state pairs. Two types of connections are used to connect these pairs. An in-state connection indicates that for a transition to fire the composite object must be in the connected state in another of its

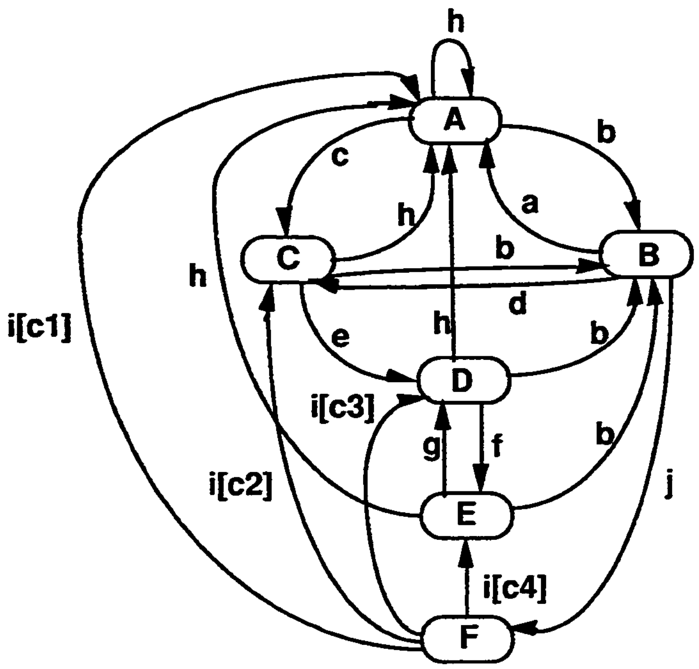


Figure 4.5: State-transition diagram.

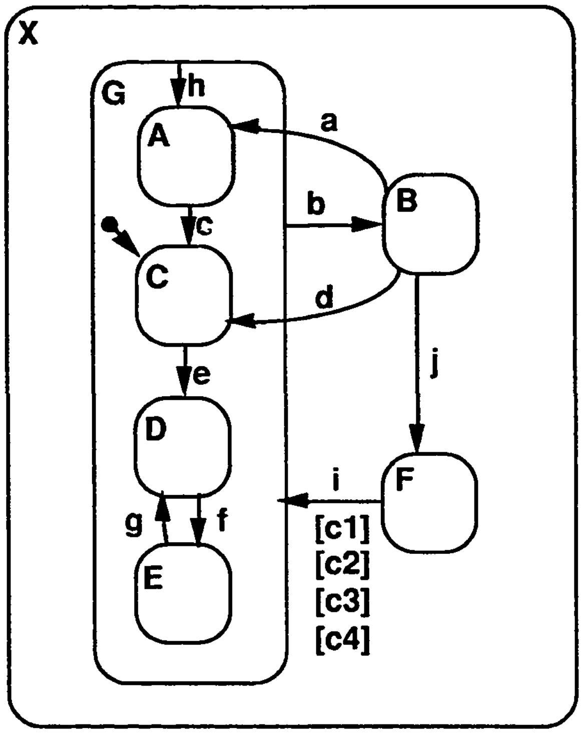


Figure 4.6: Exclusive-OR state.

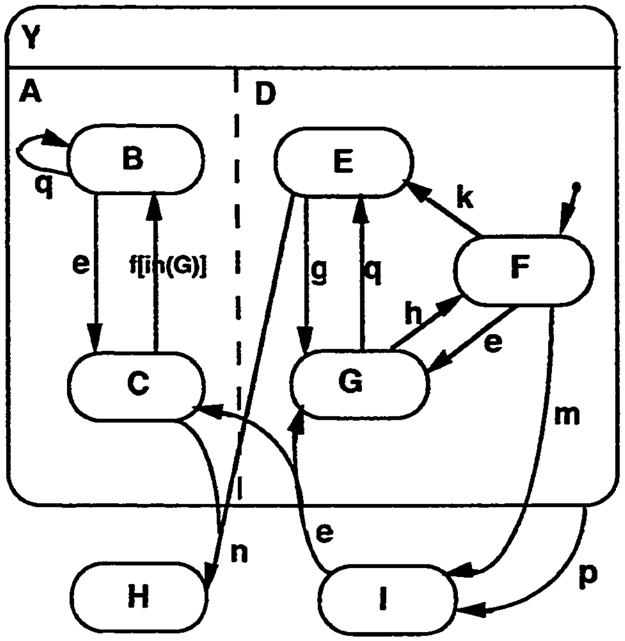


Figure 4.7: AND-state.

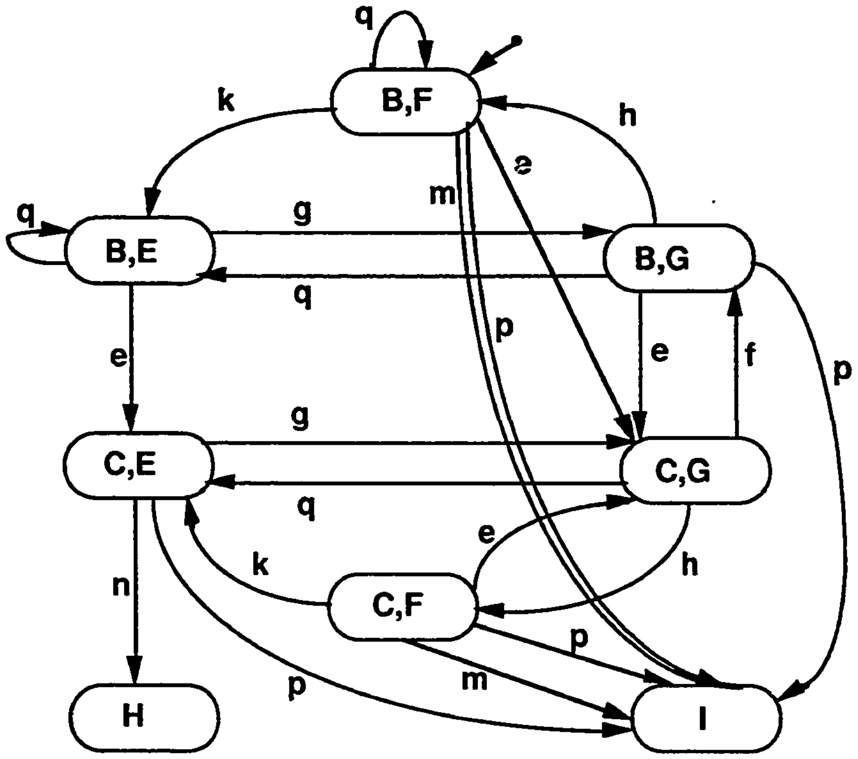


Figure 4.8: AND-state expansion

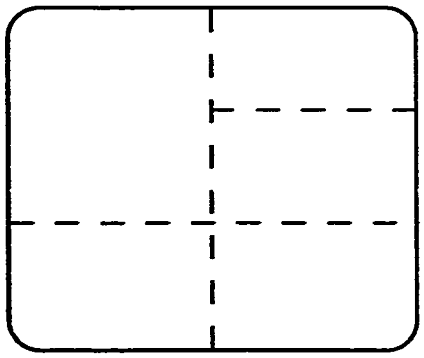


Figure 4.9: AND-state partitioning

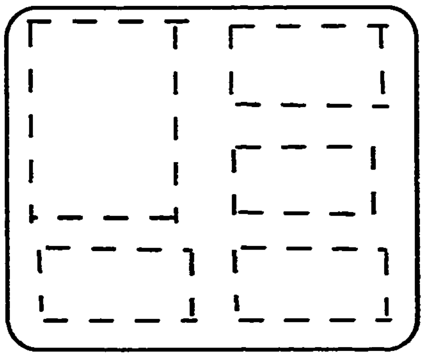


Figure 4.10: Conduits in AND-state

substates. A not-in-state connection indicates that for a transition to fire the composite object must not be in the connected state in another of its substates. An in-state connection is represented by a line ending with dots joining a transition-arc and a state as in Figure 4.11. A not-in-state connection is represented by a line ending with small circles joining a transition-arc and a state as in Figure 4.12.

Sometimes for a transition to fire, it can require a composite object to be in two or more other substates. To reduce the number of such transition-arc state connections we can join the connecting lines by a dot. We call such a dot an AND-connection. Similarly the firing of a transition can require a composite object to be in any one of two or more particular states. We use a dot surrounded by a small circle to represent an OR-connection. A guarded condition on a transition participating in a transition-arc state pair can either be an AND-condition or an OR-condition. An AND-condition means that the firing of a transition is only possible if both the AND-condition and the condition implied by the transition-arc state pair are satisfied. An OR-condition means that the firing of a transition is only possible if either the OR-condition is satisfied or the condition implied by the transition-arc state pair is satisfied. We represent an AND-condition by placing the symbol $\wedge$ in front of the guarded condition and an OR-condition by placing the symbol V. The use of these notations is shown in Figure 4.13.

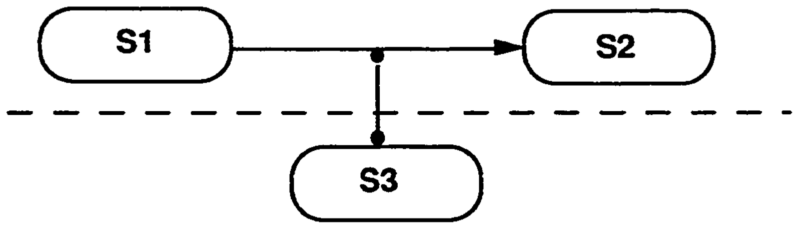


Figure 4.11: An in-state connection

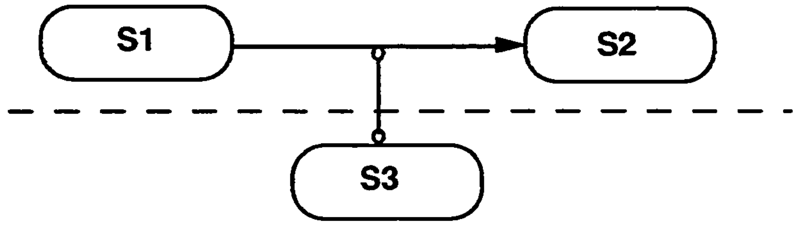


Figure 4.12: A not-in-state connection

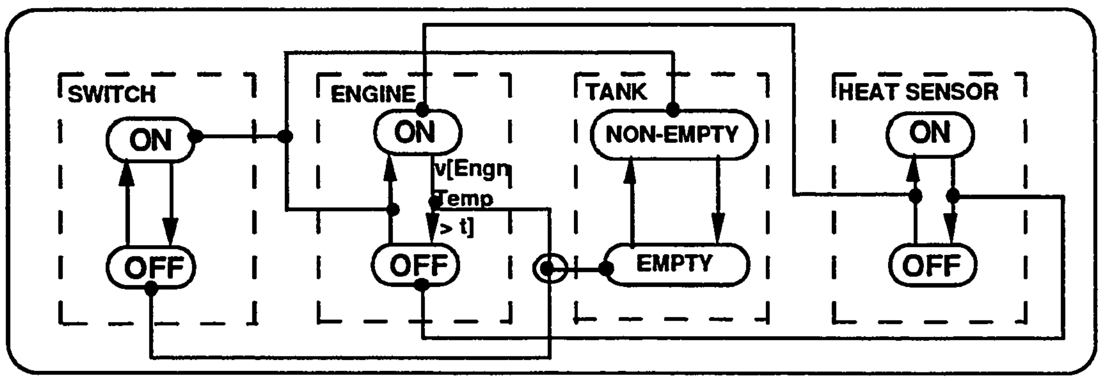


Figure 4.13: Dependent AND-states

We use the notation in Figure 4.14 for a state transition. M1 is the stimulus message triggering the transition, C1 is the class of the object sending the message. M2 is the response message sent by the object, C2 is the class of the object to which this message is sent. CONDITION represents a guard on the transition if the transition is guarded. A transition arrow without a stimulus message name indicates an automatic transition that fires when the activity associated with the source state is completed. As an alternative to showing response-messages on transitions, response messages can be associated with entering or exiting a state. If all transitions into a state result in the same response-message, the notation: entry / response-message is used inside the state box. Similarly, the notation: exit / response-message is used if all the transitions out of a state result in the same response-message. This notation is taken from Rumbaugh et al. [RBP+91].

We use the notation $\underset{\text{Event}}{\longleftarrow}$ on a transition arc to denote an event which must have occurred for that particular transition to occur.

### 4.7.3 Steps in constructing the dynamic model

1. For each event-response list formed in the exploratory stage trace the sequences of internal events and their responses generated by the external event. Record down the classes participating in the event-response sequence.
2. Obtain more events for a class from its class specification sheet. list all the messages received and sent by an object class in a given role.
3. With the help of CRC card for a class, list down the states the class can assume under each role responsibility, public as well as private.

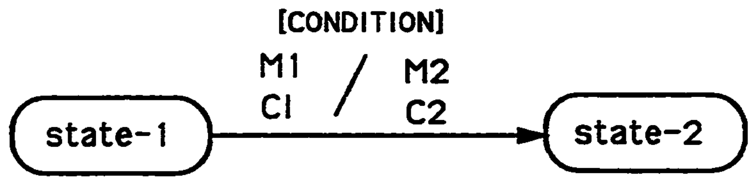


Figure 4.14: State-transition notation

Consider relevant attributes only when defining a state, pay particular attention to attributes whose value changes affect the dynamic behaviour of the class.

4. Use different what-if scenarios to generate more states.
5. Abstract the states found in step 3 into exclusive-OR states if possible.
6. For each state construct a stimulus-message response-message table as in Figure 4.15.
7. Map the stimulus events on a class to the signatures obtained for that class. Deduce missing signatures by what-if scenarios.
8. Determine the sequence of events sent by a class as a response to a stimulus event. Capture this sequence in a high-level control of flow diagram within the relevant state of the class. For each event sent by a class, specifically mention the collaborator class in the control flow diagram.
9. Combine the states obtained for each role responsibility to form the Statechart for that particular role. Each role forms an abstract exclusive-OR state.
10. Combine the statecharts of the roles to form the Statechart for the whole class.
11. For a composite class follow the above steps in drawing a Statechart for each component class. Join these statecharts into an AND-state. Find the interactions between component states and identify guarded transitions.

Note that a subclass inherits the statecharts of its superclasses. If a subclass adds a new service an extra state and transitions corresponding to the new service are added to the Statechart. If a subclass does not add new services but it redefines a service of its superclass then the Statechart is the same except that it will have some transitions with different firing conditions.

| STATE: | | | CLASS: | | |
|---|---|---|---|---|---|
| STIMULUS MESSAGE | SENDING CLASS | RESPONSE MESSAGE | RECEIVING CLASS | GUARD ON TRANSITION | NEXT STATE |
| •<br>•<br>• | | | | | |

Figure 4.15: State table

## 4.8 The integration of the design subtasks

In this final high-level design stage all classes and subsystems obtained in the various design subtasks are integrated. More design consistency checks are carried out. The responsibilities and collaborations of classes and subsystems are then validated against the requirements specification in the presence of the user. Revisions are made if necessary. Class specification sheets and subsystem cards are then finalised.

# CHAPTER 5
# DESIGN -CONSISTENCY CHECKING

## 5.1 Introduction

Object-oriented software design is an iterative process in which an application's inheritance graph is built using both generalisation and specialisation. Existing classes may be generalised into more abstract classes and abstract classes may be specialised into more concrete classes. In addition to generalisation and specialisation, a wide variety of changes to the inheritance graph may be required before the desired inheritance graph is obtained. The types of changes required for a preliminary design may include addition and deletion of classes, addition and deletion of attributes or signatures from classes, alteration of the is-a relationship between classes, and others. As these changes are performed, the structural consistency of the inheritance graph must be ensured.

In this chapter we give a number of design-consistency checking algorithms. Our work in this chapter is greatly influenced by work in object-oriented database schema evolution [BCG+87], [Bar91], [DeZ91], [Hyo91], [Kim90], [LeH90].

## 5.2 Taxonomy of Inheritance-graph modifications

There are three types of changes that can occur to the inheritance-graph of an object-oriented software at the preliminary design stage. The first is changes to the contents of a class, the second is changes to the name of a class, and the third is changes to the structure of the inheritance-graph. The following is a taxonomy of inheritance-graph changes at the preliminary design stage :

**1. Changes to the contents of a class**

- 1.1 Changes to an attribute
  - 1.1.1 Add a new attribute to a class.
  - 1.1.2 Delete an existing attribute from a class.
  - 1.1.3 Change the name of an attribute of a class.
- 1.2 Changes to a signature
  - 1.2.1 Add a new signature to a class.
  - 1.2.2 Delete an existing signature from a class.
  - 1.2.3 Change a signature.
    - 1.2.3.1 Change the name of a signature.
    - 1.2.3.2 Change the parameter list of a signature.
    - 1.2.3.3 Change the domain of the returned value of a signature if any.

**2. Change the name of a class.**

**3. Changes to the inheritance-graph structure.**

- 3.1 Make a class S a superclass/subclass of a class C.
- 3.2 Remove a class from the superclass/subclass list of a class C.
- 3.3 Add a new class.
- 3.4 Delete an existing class.
- 3.5 Change the is-a semantics between a pair of subclasses of a class.

## 5.3 Invariants of Inheritance-graph modifications

An inheritance-graph is structurally consistent if and only if any of the following invariants is not violated :

1. Structure invariant

   There are two views regarding how the inheritance-graph of an object-oriented software should be structured. One view, reflected in languages such as Smalltalk-80 [GoR83], Objective-C [Cox86], CLOS [Kee89], [Moo89], and LOOPS [Bob83], holds that the inheritance-graph should have one root only. The second view, represented by languages such as C++ [ElS90], Eiffel [Mey88], and Hybrid [Nie87], holds that the inheritance-graph can be composed of several inheritance-trees. Each inheritance-tree can in turn have several roots.

   In the design-consistency checking algorithms we shall discuss, we assume, without loss of generality, the following structure invariant : The inheritance-graph is rooted, connected, directed, non-transitive, and acyclic.

2. Unique-name invariant

   Each class in the inheritance-graph must have a unique name. All attributes of a class must have unique names. Similarly, all signatures of a class must have unique names.

3. Full-inheritance invariant

   A class inherits the union of attributes and the union of signatures from its superclasses with the exception that if the class defines an attribute or signature with the same name as that occurring in one or more of its superclasses then that attribute or signature is never inherited. If more than one superclass of a class, occurring in different inheritance-paths to the class, define an attribute or a signature with identical names and if an attribute or a signature with that name is not defined in the class, a name conflict occurs. Figure 5.1 illustrates an attribute name-conflict. Both class $C_2$ and $C_3$ define an attribute with name p resulting in a name conflict in $C_4$. The name conflict can be removed by defining p in $C_4$, in that case we say that the attribute p in $C_4$ blocks the name conflict.

4. Domain compatibility invariant

   If a class $C_2$ defines an attribute with the same name as an attribute it would otherwise inherit from its superclass $C_1$, then the domain of $C_2$'s attribute must either be the same as that of $C_1$'s attribute or it must be a subclass of the domain of $C_1$'s attribute. For example, in Figure 5.2 a class $C_1$ defines an attribute p with domain $C_3$, and $C_2$, a subclass of $C_1$, defines an attribute p with domain $C_4$. Domain compatibility invariant is preserved since $C_4$ is a subclass of $C_3$.

   We use the notation domain(p,$C_1$) $\leq$ domain(p,$C_2$) to indicate that the domain of an attribute p defined in a class $C_1$ is the same or is a subclass of the domain of an attribute p defined in class $C_2$.

5. Signature compatibility invariant

   If a class B defines a signature with the same name as a signature it would otherwise inherit from its superclass C then the signature defined in B must be signature compatible with that in C. We use the notation Signature(s,B) $\leq$ Signature(s,C) to indicate that a signature named s, defined in a class B, is signature compatible with a signature named s, defined in a class C.

$$\text{Signature}(s,B) \leq \text{Signature}(s,C) \text{ if and only if}$$
$$\text{Signature}(s,B) = s(p_1 : B_1, p_2 : B_2, \ldots, p_n : B_n) : B_{n+1},$$
$$\text{Signature}(s,C) = s(p_1 : C_1, p_2 : C_2, \ldots, p_n : C_n) : C_{n+1}$$
$$\text{and } B_i \leq C_i, \text{ for } i = 1, 2, 3, \ldots, n+1$$

   where $p_i$ is a parameter, $B_i$ and $C_i$ for i = 1, 2, 3, ... ,n are parameter domain classes and $B_{n+1}$ and $C_{n+1}$, if present, are the domains of the objects returned by the signatures.

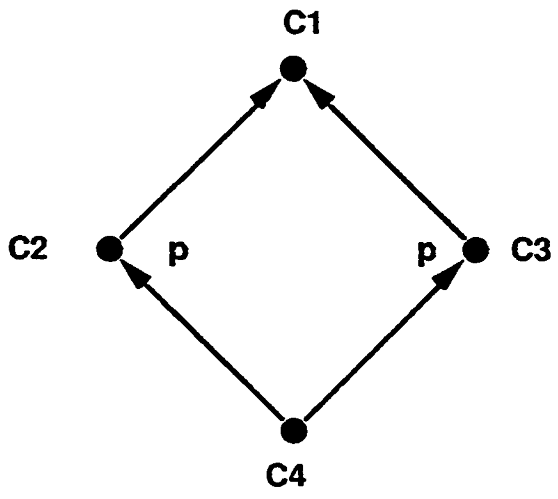


Figure 5.1: Attribute name conflict

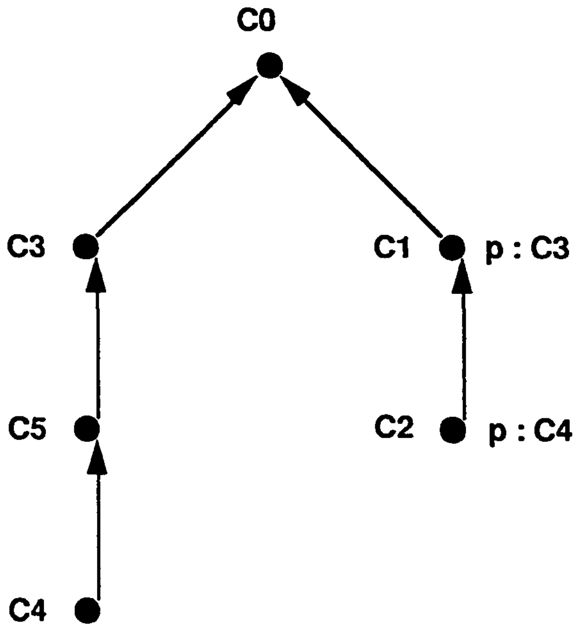


Figure 5.2: Domain Compatibility

6. Attribute-domain invariant

   The domain of an attribute, if it is not predefined, must be a class in the inheritance-graph.

7. Parameter-domain invariant

   The domain of a parameter, if it is not predefined, must be a class in the inheritance-graph.

8. Returned-object-domain invariant

   The domain of the object returned by a signature, if it is not predefined, must be a class in the inheritance-graph.

9. Is-a semantics invariant

   Two classes specified as having exclusive is-a semantics cannot have a common subclass.

10. Is-a semantics compatibility invariant

    If classes $B_1$ and $C_1$ have no is-a relationship then two classes B and C having no is-a relationship and having both $B_1$ and $C_1$ as either direct- or indirect-superclasses cannot have different is-a semantics in $B_1$ and $C_1$. For example, in Figure 5.3 the is-a semantics for $C_{11}$ and $C_{12}$ defined in $C_1$, i.e the is-a semantics for $C_3$ and $C_4$, must be the same as the is-a semantics for $C_{11}$ and $C_{12}$ defined in $C_2$, i.e the is-a semantics for $C_5$ and $C_6$.

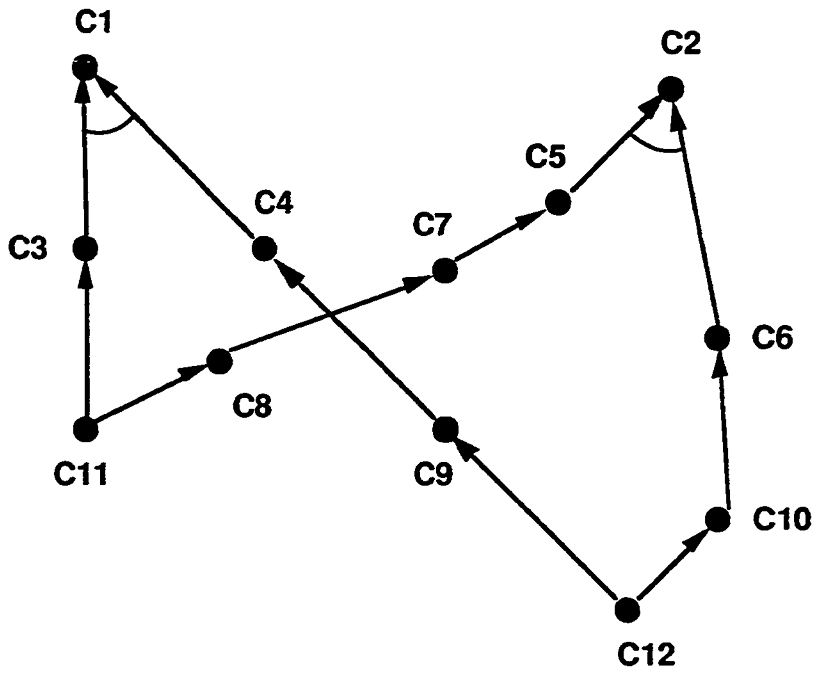


**Figure 5.3: Is-a semantics compatibility**

## 5.4 Design-consistency checking algorithms

### 5.4.1 Introduction

As mentioned in section 5.2, the process of object-oriented software design is an iterative one. The final design is only obtained after making a wide variety of changes to the inheritance-graph being designed. Some of these changes may violate the structural consistency of the graph by violating one or more invariants given in the previous section .

To ensure that inheritance-graph modifications do not violate the structural-consistency of the graph, we give a number of design-consistency checking algorithms. We start with six definitions which are important in the discourse to follow.

**Definition: Attribute-scope**

If an attribute p is inherited by or defined in a class C, then the scope of that attribute for the class C, Attribute-Scope(p, C), is the class C and all subclasses of C where the attribute is inherited from C, with possible name conflicts. For example, in Figure 5.4 p and q are attributes defined in classes as shown. Attribute-Scope(p, $C_1$) = {$C_1$, $C_3$, $C_4$, $C_7$, $C_{10}$}, Attribute-Scope(q, $C_1$) = {$C_1$, $C_2$, $C_5$, $C_8$}.

**Definition: Signature-scope**

If a signature s is inherited by or defined in a class C, then the scope of that signature for the class C, Signature-Scope(s, C), is the class C and all subclasses of C where the signature is inherited from C, with possible name conflicts.

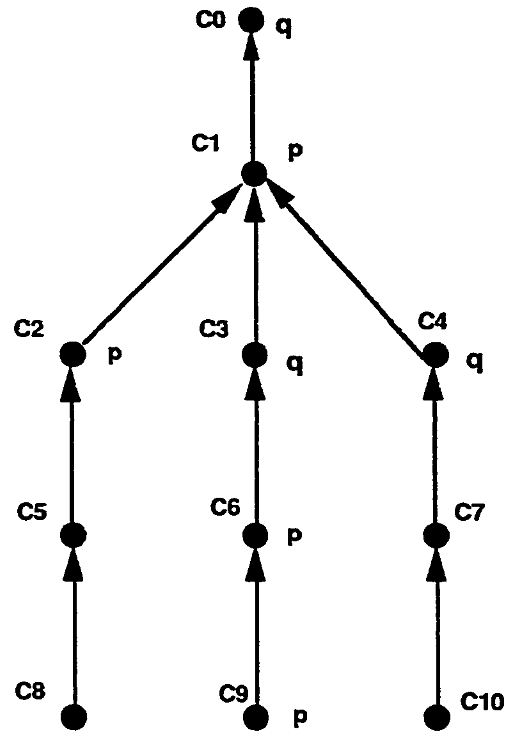


**Figure 5.4: Attribute-scope**

**Definition: Attribute positive-shield**

If an attribute p is inherited by or defined in a class C, then the positive-shield of that attribute for the class C, Attribute-Positive-Shield(p, C), are those subclasses of C in which p is redefined and which would have inherited p from C if not for the redefinitions. For example, in Figure 5.4 Attribute-Positive-Shield(p, $C_1$) = {$C_2$, $C_6$}, Attribute-Positive-Shield(q, $C_1$) = {$C_3$, $C_4$}.

**Definition: Signature positive-shield**

If a signature s is inherited by or defined in a class C, then the positive-shield of that signature for the class C, Signature-Positive-Shield(s, C), are those subclasses of C in which s is redefined and which would have inherited s from C if not for the redefinitions.

**Definition: Attribute negative-shield**

If an attribute p is defined in a class C then the negative shield of that attribute for the class C, Attribute-Negative-Shield(p, C), is the set of all superclasses of C in which p is defined and from which C would have inherited p, with possible name conflicts, if p had not been redefined in C. For example, in Figure 5.5, Attribute-Negative-Shield(p, $C_9$) = {$C_3$, $C_4$, $C_8$}, Attribute-Negative-Shield(q, $C_9$) = {$C_5$}.

**Definition: Signature negative-shield**

If a signature s is defined in a class C then the negative shield of that signature for the class C, Signature-Negative-Shield(s, C), is the set of all superclasses of C in which s is defined and from which C would have inherited s, with possible name conflicts, if s had not been redefined in C.

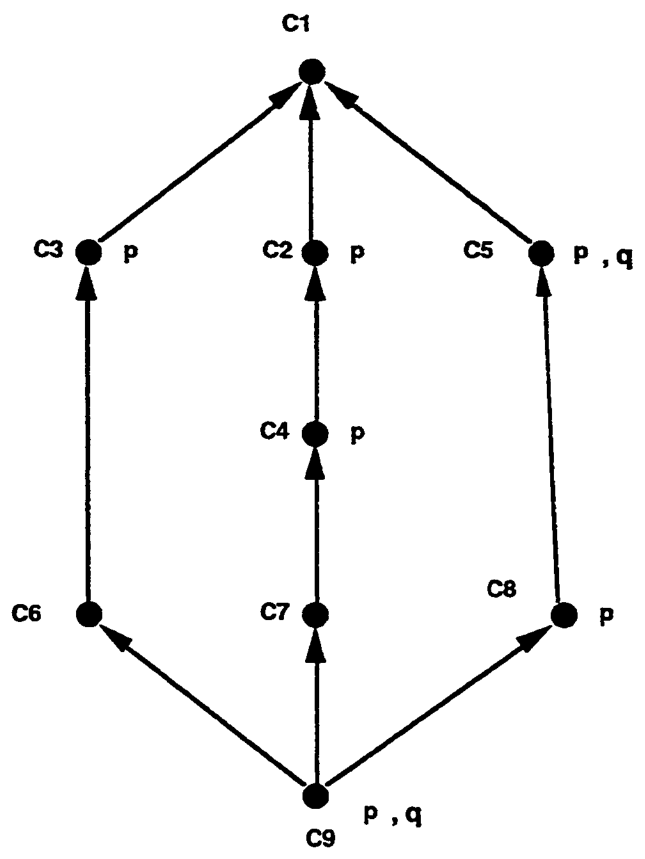


Figure 5.5: Attribute Negative Shield

## 5.4.2 Inheritance-graph modifications

### 1. Adding an attribute to a class

In order to add an attribute p to a class C we have to ensure that:

1. The unique-name invariant is not violated.
2. The attribute-domain invariant is not violated.
3. The domain-compatibility invariant is not violated:

   This involves: (i) The determination of whether the added attribute is domain-compatible with an attribute or attributes with a similar name, if any, defined in Attribute-Negative-Shield(p, C). We call this an upward-domain-compatibiliy-check. (ii) The determination of whether the added attribute is domain-compatible with an attribute or attributes with a similar name, if any, defined in Attribute-Positive-Shield(p, C). We call this a downward-domain-compatibiliy-check.

   For example, in Figure 5.6, an attribute p with domain $C_1$ is added to class $C_3$. The added attribute causes upward-domain-incompatibility since domain(p, $C_3$) $\nleq$ domain(p, $C_2$). In Figure 5.7, an attribute p with domain $C_2$ is added to class $C_3$. Although the added attribute does not cause downward-domain-incompatibility in $C_5$ since domain(p, $C_5$) $\leq$ domain(p, $C_3$), it causes downward-domain-incompatibility in $C_6$ since domain(p, $C_6$) $\nleq$ domain(p, $C_3$).

4. The full-inheritance invariant is not violated:

   This involves the determination of whether the added attribute causes name-conflicts in Attribute-Scope(p, C). These will occur if

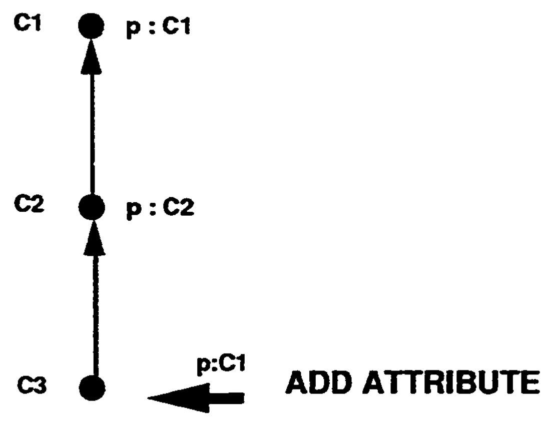


Figure 5.6: Upward-domain-incompatibility

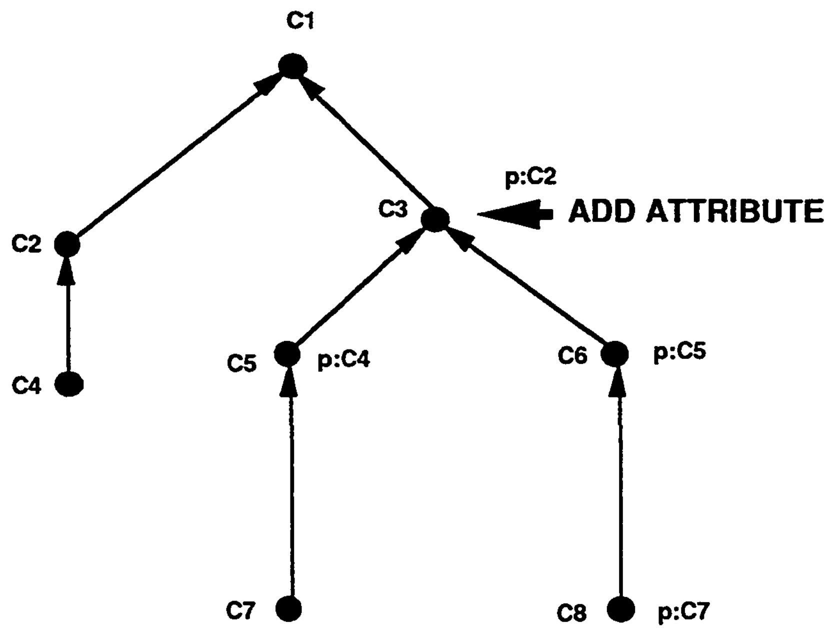


Figure 5.7: Downward-domain-incompatibility

Attribute-Scope(p, C) $\cap$ Attribute-Scope(p, $C_i$) $\neq \emptyset$ , where $C_i$ is a class in which p is defined and $C_i \notin$ Subclasses(C). For example, in Figure 5.8 the addition of an attribute p in $C_2$ causes name conflicts in $C_4$ since Attribute-Scope(p, $C_2$)$\cap$ Attribute-Scope(p, $C_1$) = $\{C_2, C_4\} \cap \{C_1, C_3, C_4, C_5\} = \{C_4\} \neq \emptyset$.

The algorithms for determining whether or not Unique-name invariant and Attribute-domain invariant are violated are trivial. To check for domain-compatibility and name-conflicts, we define the following algorithms :

**Algorithm 1: Upward domain-compatibility checking algorithm**

```
function UPWARD-DOMAIN-COMPATIBLE(p, C) : BOOLEAN
begin
    for each $C_j \in$ Attribute-Negative-Shield(p, C) do
        if domain(p, C) $\nleq$ domain(p, $C_j$) then
            return FALSE;
        endif
    endfor
    return TRUE;
end
```

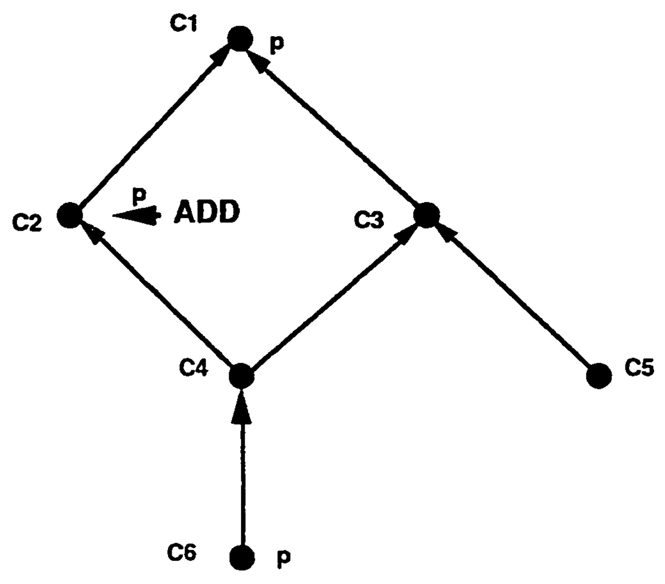


**Figure 5.8: Attribute name conflict after attribute addition**

The detailed algorithm is:

```
function UPWARD-DOMAIN-COMPATIBLE(p, C) : BOOLEAN
begin
    QUEUE := $\emptyset$ ; VISITED := $\emptyset$;
    QUEUE := QUEUE $\cup$ {C}; VISITED :=VISITED $\cup$ {C};
    while QUEUE $\neq$ $\emptyset$ do
        QUEUE := QUEUE - {$C_i$}; /* delete $C_i$ from QUEUE */
        for each direct-superclass $C_j$ of $C_i$ do
            if $C_j$ $\notin$ VISITED then
                VISITED := VISITED $\cup$ {$C_j$};
                if p is defined in $C_j$ then
                    if domain(p, C) $\nleq$ domain(p, $C_j$) then
                        return FALSE;
                    endif
                else
                    QUEUE := QUEUE $\cup${$C_j$};
                endif
            endif
        endfor
    endwhile
    return TRUE;
end
```

**Algorithm 2: Downward domain-compatibility checking algorithm**

```
function DOWNWARD-DOMAIN-COMPATIBLE(p, C) : BOOLEAN
 begin
     for each $C_j \in$ Attribute-Positive-Shield(p, C) do
         if domain(p, $C_j$) $\not\leq$ domain(p, C) then
             return FALSE;
         endif
     endfor
     return TRUE;
 end
```

The detailed algorithm can be defined in a similar way as done in Algorithm 1.

**Algorithm 3:** Name-conflict detection algorithm after attribute addition.

```
 function ATTRIBUTE-NAME-CONFLICT(p, C) : BOOLEAN
 begin
     for each Cj $\notin$ Subclasses(C) and in which p is defined do
         if Attribute-Scope(p, C) $\cap$ Attribute-Scope(p, Cj) $\neq \varnothing$ then
             return TRUE;
         endif
     endfor
     return FALSE;
 end
```

The detailed algorithm is:

```
function ATTRIBUTE-NAME-CONFLICT(p, C) : BOOLEAN
begin
    G := ATTRIBUTE-SCOPE(p, C);
    H := Subclasses(C);
    QUEUE := $\varnothing$ ; VISITED := $\varnothing$;
    QUEUE := QUEUE $\cup$ {RootClass}; VISITED := VISITED $\cup$ {RootClass};
    while QUEUE $\neq \varnothing$ do
        QUEUE := QUEUE - {$C_i$}; /* delete Ci from QUEUE */
          for each direct-subclass $C_j$ of $C_i$ do
            if $C_j \notin$ H then
              if $C_j \notin$ VISITED then
                  VISITED := VISITED $\cup$ {$C_j$};
                  QUEUE := QUEUE $\cup$ {$C_j$};
                  if p is defined in $C_j$ then
                      if G $\cap$ ATTRIBUTE-SCOPE(p, $C_j$) $\neq \varnothing$ then
                          return TRUE;
                      endif
                  endif
              endif
            endif
        endfor
    endwhile
    return FALSE;
end
```

where ATTRIBUTE-SCOPE(p, C) is the function :

```
function ATTRIBUTE-SCOPE(p, C) : SET
begin
    QUEUE := $\emptyset$ ; VISITED := $\emptyset$; SET := $\emptyset$;
    QUEUE := QUEUE $\cup$ {C}; VISITED := VISITED $\cup$ {C}: SET := SET $\cup$ {C};
    while QUEUE $\neq \emptyset$ do
        QUEUE := QUEUE - {$C_i$}; /* delete $C_i$ from QUEUE */
        for each direct-subclass $C_j$ of $C_i$ do
            if $C_j \notin$ VISITED then
                VISITED := VISITED $\cup$ {$C_j$};
                if p is not defined in $C_j$ then
                    QUEUE := QUEUE $\cup$ {$C_j$};
                    SET := SET $\cup$ {$C_j$};
                endif
            endif
        endfor
    endwhile
    return SET;
end
```

## 2. Adding a signature to a class

In order to add a signature to a class we have to ensure that :

1. The unique-name invariant is not violated.
2. The parameter-domain invariant is not violated.
3. The returned-object domain invariant is not violated.

4. The signature-compatibility invariant is not violated.

5. The full-inheritance invariant is not violated.

The algorithms for case 1, 2, and 3 are trivial, for case 4 and 5 we define the following algorithms:

**Algorithm 4: Upward signature-compatibility checking algorithm**

```
function UPWARD-SIGNATURE-COMPATIBLE(s, C) : BOOLEAN
begin
    for each $C_j \in$ Signature-Negative-Shield(s, C) do
        if signature(p, C) $\not\leq$ signature(s, $C_j$) then
            return FALSE;
        endif
    endfor
    return TRUE;
end
```

**Algorithm 5: Downward signature-compatibility checking algorithm**

```
function DOWNWARD-SIGNATURE-COMPATIBLE(s, C) : BOOLEAN
begin
    for each $C_j \in$ Signature-Positive-Shield(s, C) do
        if signature(s, $C_j$) $\not\leq$ signature(s, C) then
            return FALSE;
        endif
    endfor
    return TRUE;
end
```

**Algorithm 6:** Name-conflict detection algorithm after signature addition.

```
function SIGNATURE-NAME-CONFLICT(s, C) : BOOLEAN
begin
    for each Cj $\notin$ Subclasses(C) and in which s is defined do
        if Signature-Scope(s, C) $\cap$ Signature-Scope(s, Cj) $\neq \emptyset$ then
            return TRUE;
        endif
    endfor
    return FALSE;
end
```

## 3. Changing the name of an attribute or a signature

This modification is equivalent to adding a new attribute or a new signature to a class except that there is no need to check for the existence of the domain classes in the inheritance-graph.

## 4. Changing the domain of an attribute, parameter, or returned-object

We have to ensure that:

1. Attribute-domain invariant, parameter-domain invariant, or returned-object domain invariant is not violated, respectively.
2. Domain- or Signature-compatibility invariant is not violated.

The algorithms for case 1 are trivial. The algorithms used for case 2 are UPWARD-DOMAIN-COMPATIBLE (Algorithm 1) and DOWNWARD-DOMAIN-COMPATIBLE (Algorithm 2), if the domain of an attribute is changed, or UPWARD-SIGNATURE-COMPATIBLE (Algorithm 4) and DOWNWARD-SIGNATURE-COMPATIBLE (Algorithm 5), if the domain of a parameter or returned-object is changed.

As an example, in Figure 5.9 changing the domain of p in $C_5$ from $C_4$ to $C_1$ causes an upward-domain-incompatibility, since by doing so domain(p, $C_5$) $\nleq$ domain(p, $C_3$).

## 5. Deleting an attribute or a signature from a class

The deletion of an attribute p or a signature s from a class C will cause a name-conflict if that attribute or signature was blocking the name-conflict, i.e. if the cardinality of Attribute-Negative-Shield(p, C) is greater than one or if the cardinality of Signature-Negative-Shield(s, C) is greater than one, respectively. For example, in Figure 5.9 the deletion of attribute p from $C_6$ will cause a name-conflict in $C_6$, since the cardinality of Attribute-Negative-Shield(p, $C_6$) is 2.

**Algorithm 7**: Name-conflict detection algorithm after attribute deletion

```
function NAME-CONFLICT-IN-ATTRIBUTE-DELETION(p, C): BOOLEAN
begin
   if Cardinality( Attribute-Negative-Shield(p, C)) > 1 then
      return TRUE;
   else
      return FALSE;
   endif
end
```

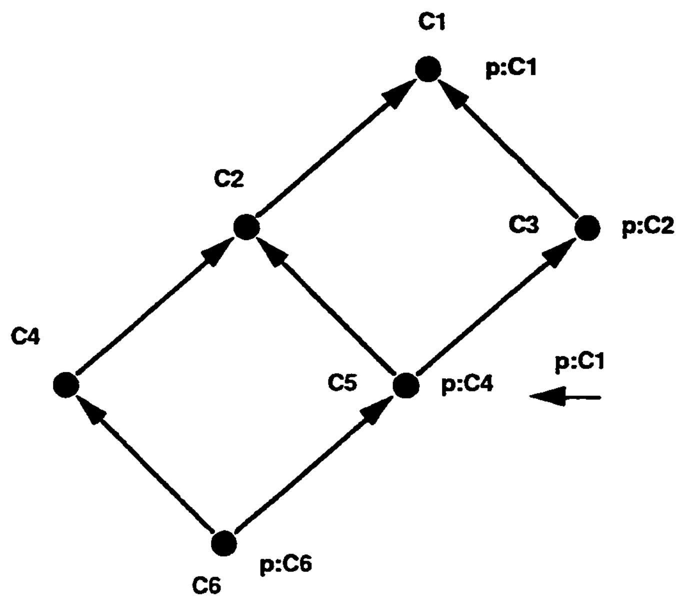


Figure 5.9: Changing an attribute's domain

The detailed algorithm is :

```
function NAME-CONFLICT-IN-ATTRIBUTE-DELETION(p, C): BOOLEAN
begin
    CARDINALITY := 0; QUEUE := $\emptyset$; VISITED := $\emptyset$;
    QUEUE := QUEUE $\cup$ {C}: VISITED := VISITED $\cup$ {C};
    while QUEUE $\neq$ $\emptyset$ do
        QUEUE := QUEUE - {$C_i$}; /* delete $C_i$ from QUEUE */
        for each direct-superclass $C_j$ of $C_i$ do
            if $C_j \notin$ VISITED then
                VISITED := VISITED $\cup$ {$C_j$};
                if p is defined in $C_j$ then
                    CARDINALITY := CARDINALITY + 1;
                    if CARDINALITY > 1 then
                        return TRUE;
                    endif
                else
                    QUEUE := QUEUE $\cup$ {$C_j$};
                endif
            endif
        endfor
    endwhile
    return FALSE;
end
```

**Algorithm 8**: Name-conflict detection algorithm after signature deletion

```
function NAME-CONFLICT-IN-SIGNATURE-DELETION(s, C): BOOLEAN
begin
   if Cardinality( Signature-Negative-Shield(s, C)) > 1 then
      return TRUE;
   else
      return FALSE;
   endif
end
```

## 6. Changing the name of a class

In order to change the name of a class we have to:

1. Check that the inheritance-graph does not have a class with a name similar to the new name, i.e. we have to ensure that the unique-name invariant is not violated.
2. Find all references to the class and change the name in those references to the new name, i.e. we have to ensure that Attribute-domain invariant, Parameter-domain invariant, and Returned-object domain invariant are not violated.

The algorithms for both cases are trivial.

## 7. Deleting a class from the Inheritance graph

There are two cases to consider :

1. The deleted class C is deleted together with all its subclasses which are descendants of C only.

2. C is deleted, but none of its subclasses is deleted. Each direct-subclass of C is made a direct-subclass of each direct-superclass of C.

In both cases we have to take into account classes in which each deleted class is referenced as a domain; appropriate modifications have to be made to these classes.

In case 1 mentioned above no possible name-conflicts can arise in deleting C, even if C was blocking name-conflicts before its deletion. In case 2, if C is blocking a name-conflict by redefining an attribute p or a signature s then the deletion of C will cause name conflicts if p or s is not redefined in each direct-subclass of C. For example in Figure 5.10a class C is blocking name conflicts by redefining an attribute p and a signature s. If C is deleted, resulting in Figure 5.10b, no attribute-name conflicts occur since p is redefined in each former direct-subclass of C. However, a signature-name conflict occurs in $C_7$ and $C_8$ .

The algorithm for determining whether the deletion of a class C will cause name-conflicts is then :

**Algorithm 9:** Name-conflict detection algorithm after a class deletion

1. Find all attributes and signatures whose name-conflicts are blocked by C, if any.
2. For each direct-subclass of C determine whether each of the attributes and signatures found in step 1 is redefined. If not, a name conflict will occur in deleting C.

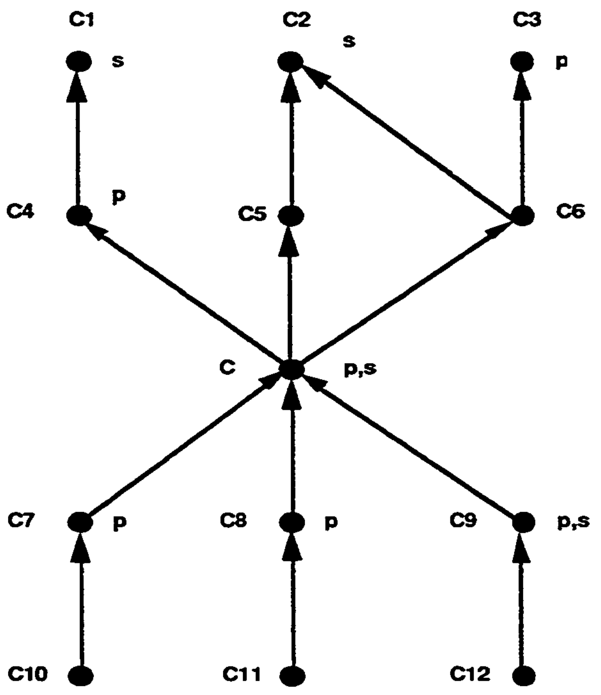


Figure 5.10a: A class blocking name conflicts

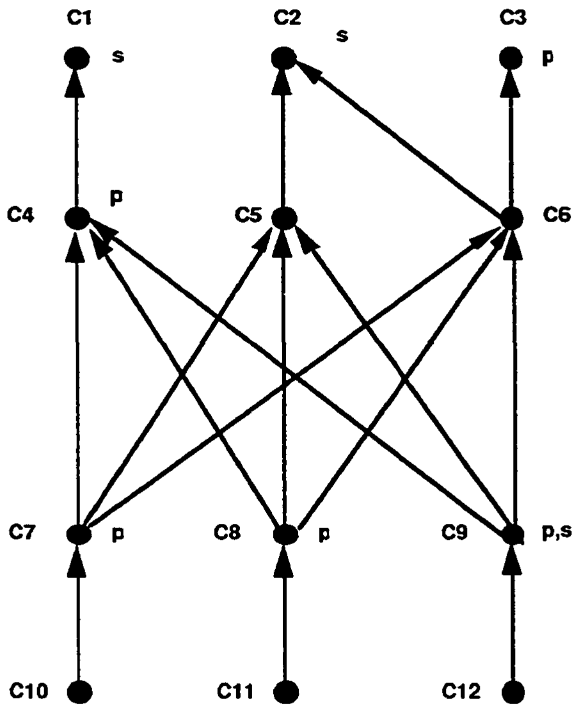


Figure 5.10b: Name-conflict in class deletion

The detailed algorithm makes use of the functions NAME-CONFLICT-IN-ATTRIBUTE-DELETION (Algorithm 7) and NAME-CONFLICT-IN-SIGNATURE-DELETION (Algorithm 8):

```
function NAME-CONFLICT-IN-CLASS-DELETION(C) : BOOLEAN
begin
    for each attribute p of C do
        if NAME-CONFLICT-IN-ATTRIBUTE-DELETION(p, C) then
            for each direct-subclass $C_i$ of C do
                if p is not defined in $C_i$ then
                    return TRUE;
                endif
            endfor
        endif
    endfor
    for each signature s of C do
        if NAME-CONFLICT-IN-SIGNATURE-DELETION(s, C) then
            for each direct-subclass $C_i$ of C do
                if s is not defined in $C_i$ then
                    return TRUE;
                endif
            endfor
        endif
    endfor
    return FALSE;
end
```

## 8. Adding a new class to the Inheritance-graph

We consider two basic cases:

1. Adding a new class C as a direct-superclass to a specified class $C_i$.
2. Adding a new class C as a direct-subclass to a specified class $C_i$.

Adding a class as a direct-superclass/direct-subclass to a number of specified classes is equivalent to adding the new class as a direct-superclass or direct-subclass to only one of the classes, then adding inheritance-links to the rest of the classes, one at a time, with the deletion of any redundant inheritance links which may arise. We defer the discussion of inheritance-link deletion and addition.

In adding a new class C as a direct-superclass or as a direct-subclass to a specified class $C_i$ we must ensure that:

1. The new class does not have local name-conflicts.
2. The inheritance-graph does not have a class having the same name as the new class.
3. The specified class $C_i$ exists in the inheritance-graph.
4. Each attribute, parameter, or returned-object of the new class, if it is not predefined, has as its domain a class in the inheritance-graph.

If the new class is added as a direct-superclass we must in addition ensure that:

5. The new class is made a direct-subclass of the root class (since by assumption the inheritance-graph must have a single root).
6. Each attribute p of the new class is downward-domain-compatible with the definitions of p in Attribute-Positive-Shield(p, C), if the shield is not empty.

7. Each signature s of the new class is downward-signature-compatible with the definitions of s in Signature-Positive-Shield(s, C), if the shield is not empty.
8. Each attribute p and each signature s of the new class does not introduce name-conflicts in Attribute-Scope(p, C) and Signature-Scope(s, C), respectively.

If the new class is added as a direct-subclass we must in addition ensure that:

9. Each attribute p of the new class is upward-domain-compatible with the definitions of p in Attribute-Negative-Shield(p, C), if the shield is not empty.
10. Each signature s of the new class is upward-signature-compatible with the definitions of s in Signature-Negative-Shield(s, C), if the shield is not empty.

The algorithms for case 1 to 5 are trivial. For case 6 to 10 we use the following algorithm, which makes use of algorithms 1, 2, 3, 4, 5, and 6:

**Algorithm 10: Conflict detection algorithm after a class addition**

```
function ADDED-CLASS-IS-INHERITANCE-GRAPH-COMPATIBLE(C) : BOOLEAN
begin
    for each attribute p of C do
        if not UPWARD-DOMAIN-COMPATIBLE(p, C) then
            return FALSE;
        endif
        if not DOWNWARD-DOMAIN-COMPATIBLE(p, C) then
            return FALSE;
        endif
        if ATTRIBUTE-NAME-CONFLICT(p, C) then
            return FALSE;
        endif
    endfor
    for each signature s of C do
        if not UPWARD-SIGNATURE-COMPATIBLE(s, C) then
            return FALSE;
        endif
        if not DOWNWARD-SIGNATURE-COMPATIBLE(s, C) then
            return FALSE;
        endif
        if SIGNATURE-NAME-CONFLICT(p, C) then
            return FALSE;
        endif
    endfor
    return TRUE;
end
```

## 9. Deleting an Inheritance-link

In deleting an inheritance-link we must ensure that:

1. A class is never deleted from the inheritance-graph (a class can only be deleted by a delete-class operation).
2. Domain-compatibility and signature-compatibility invariants are not violated. For example, in Figure 5.11 the deletion of the inheritance-link between class $C_6$ and $C_8$ will result in domain-compatibility invariant violation since in that case domain(q, $C_7$) $\not\leq$ domain(q, $C_4$) and domain(p, $C_{10}$) $\not\leq$ domain(p, $C_9$).

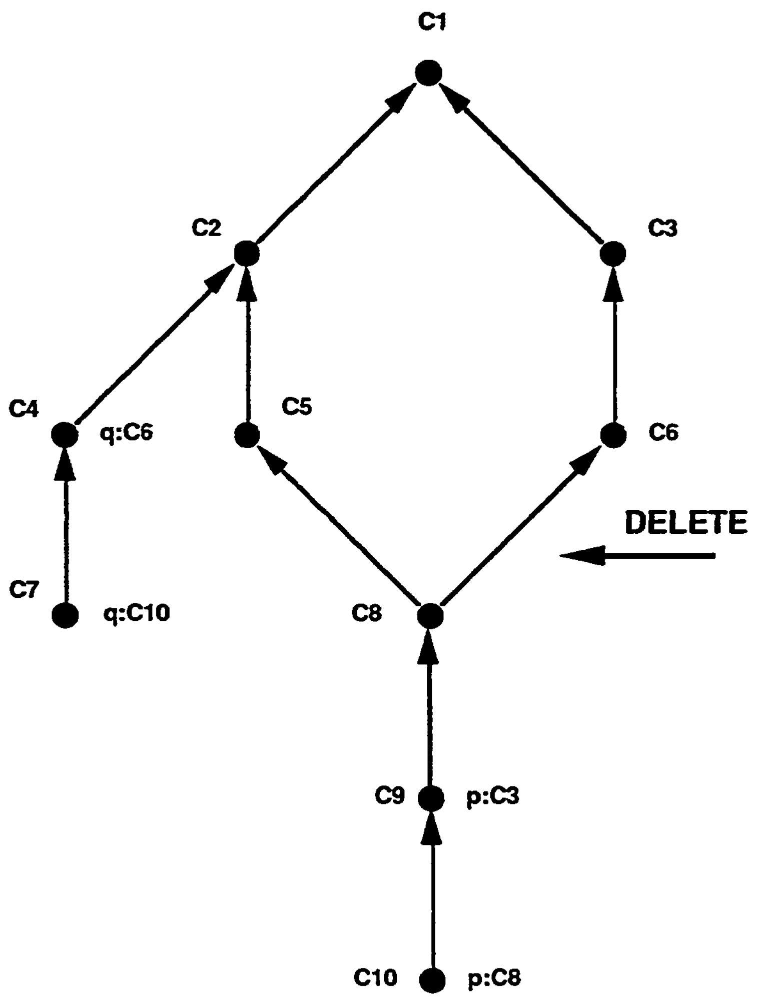


Figure 5.11: Inheritance-link deletion

**Algorithm 11: Domain- and Signature-compatibility checking algorithm after an inheritance-link deletion**

Let Parameter(s, C) be a Boolean function which returns true if a signature s has one or more parameters with domains in C. Let ReturnedObject(s, C) be a Boolean function which returns true if a signature s returns an object with domain in C. Let DeleteLink(C, $C_i$) be a procedure which deletes the inheritance-link between a class $C_i$ and its direct-subclass C and which will join C to the root class if C has only $C_i$ as a direct-superclass. Let AddLink(C, $C_i$) be a procedure which makes C a direct-subclass of $C_i$, deleting any direct-link between C and the root class if such a link exists. The algorithm for determining whether the deletion of an inheritance-link will cause domain- or signature-compatibility invariant violation is then :

```
function INCOMPATIBILITY-IN-DELETING-A-LINK(C, $C_i$) : BOOLEAN
begin
    S1 := Superclasses(C);
    DeleteLink(C, $C_i$);
    S2 := Superclasses(C);
    G := S1 - S2;
    for each class $C_j$ do
        for each attribute p of $C_j$ do
            if domain(p, $C_j$) $\in$ G then
                if not DOWNWARD-DOMAIN-COMPATIBLE(p, $C_j$) then
                    AddLink(C, $C_i$);
                    return TRUE;
                endif
            endif
        endfor
```

```
            for each signature s of $C_j$ do
               if Parameter(s, G) or ReturnedObject(s, G) then
                  if not DOWNWARD-SIGNATURE-COMPATIBLE(s, $C_j$) then
                     AddLink(C, $C_i$);
                     return TRUE;
                  endif
               endif
            endfor
        endfor
end
```

## 10. Adding an Inheritance-link

The addition of an inheritance-link is accepted if :

1. The link does not introduce a cycle.
2. The link is not redundant.
3. The link does not violate is-a semantics.
4. The link does not introduce is-a semantics incompatibilities.
5. The link does not introduce domain- or signature-incompatibilities.
6. The link does not introduce attribute- or signature-name conflicts.

**Algorithm 12: Cycle detection algorithm**

Assuming that the inheritance-graph has no cycle before the addition of the link, the cycle detection algorithm is :

1. Add the inheritance-link.
2. Perform a superclass breadth-first search starting at C, the subclass at the end of the added link. If C is found in the search then a cycle exists otherwise it does not.

The detailed algorithm is given as the Boolean function A-CYCLE-EXISTS :

```
function A-CYCLE-EXISTS(C, $C_i$): BOOLEAN
begin
    AddLink(C, $C_i$); QUEUE := $\emptyset$; VISITED := $\emptyset$;
    QUEUE := QUEUE $\cup$ {C}; VISITED := VISITED $\cup$ {C};
    while QUEUE $\neq$ $\emptyset$ do
        QUEUE := QUEUE - {$C_j$}; /* delete $C_j$ from QUEUE */
        for each direct-superclass $C_k$ of $C_j$ do
            if $C_k \notin$ VISITED then
                VISITED := VISITED $\cup$ {$C_k$};
                if $C_k$ = C then
                    DeleteLink(C, $C_i$);
                    return TRUE;
                else
                    QUEUE := QUEUE $\cup$ {$C_k$};
                endif
            endif
        endfor
    endwhile
    return FALSE;
end
```

**Algorithm 13: Link redundancy detection algorithm**

Assuming that the inheritance-graph is consistent before the addition of the link, the link redundancy detection algorithm is :

Perform a superclass breadth-first search starting at C, the subclass at the tail of the intended link, before the link is added. If $C_i$, the superclass at the head of the intended link, is found then the intended link is redundant otherwise it is not.

The detailed algorithm is given as the Boolean function ADDED-LINK-IS-REDUNDANT :

```
function ADDED-LINK-IS-REDUNDANT(C, $C_i$): BOOLEAN
begin
    QUEUE := $\varnothing$; VISITED := $\varnothing$;
    QUEUE := QUEUE $\cup$ {C}; VISITED := VISITED $\cup$ {C};
    while QUEUE $\neq \varnothing$ do
        QUEUE := QUEUE - {$C_j$}; /* delete $C_j$ from QUEUE */
        for each direct-superclass $C_k$ of $C_j$ do
            if $C_k \notin$ VISITED then
                VISITED := VISITED $\cup$ {$C_k$};
                if $C_k$ = Ci then
                    return TRUE;
                else
                    QUEUE := QUEUE $\cup$ {$C_k$};
                endif
            endif
        endfor
    endwhile
    return FALSE;
end
```

**Algorithm 14: Conflict detection algorithm for an added inheritance-link**

To ensure that an inheritance-link added to make C a direct-subclass of $C_i$ neither introduces domain- or signature-incompatibilities nor does it introduce attribute- or signature-name conflicts we use the algorithm:

```
function A-CONFLICT-EXISTS-IN-ADDING-A-LINK(C, $C_i$): BOOLEAN
begin
    AddLink(C, $C_i$);
    for each attribute p defined in or inherited by $C_i$ do
        if p introduces downward-domain-incompatibility then
            DeleteLink(C, $C_i$);
            return TRUE;
        endif
        if p introduces an attribute-name conflict then
            DeleteLink(C, $C_i$);
            return TRUE;
        endif
    endfor
    for each signature s defined in or inherited by $C_i$ do
        if s introduces downward-signature-incompatibility then
            DeleteLink(C, $C_i$);
            return TRUE;
        endif
        if s introduces a signature-name conflict then
            DeleteLink(C, $C_i$);
            return TRUE;
        endif
    endfor
    return FALSE;
end
```

The detailed algorithm is:

```
function A-CONFLICT-EXISTS-IN-ADDING-A-LINK(C, $C_i$): BOOLEAN
begin
    QUEUE := $\emptyset$ ; VISITED := $\emptyset$ ; ATTRIBUTE-LIST := $\emptyset$ ; SIGNATURE-LIST := $\emptyset$ ;
    QUEUE := QUEUE $\cup\{C_i\}$; VISITED := VISITED $\cup\{C_i\}$;
    AddLink(C, $C_i$);
    while QUEUE $\neq \emptyset$ do
        QUEUE := QUEUE - $\{C_j\}$;  /* delete $C_j$ from QUEUE */
        for each attribute p of $C_j$ do
           if p $\notin$ ATTRIBUTE-LIST then
               ATTRIBUTE-LIST := ATTRIBUTE-LIST $\cup\{p\}$;
               if not DOWNWARD-DOMAIN-COMPATIBLE(p, $C_i$) then
                   DeleteLink(C, $C_i$);
                   return TRUE;
               endif
               if ATTRIBUTE-NAME-CONFLICT(p, $C_i$) then
                   DeleteLink(C, $C_i$);
                   return TRUE;
               endif
           endif
        endfor
        for each signature s of $C_j$ do
          if s $\notin$ SIGNATURE-LIST then
              SIGNATURE-LIST := SIGNATURE-LIST $\cup$ {s};
              if not DOWNWARD-SIGNATURE-COMPATIBLE(s, $C_i$) then
                  DeleteLink(C, $C_i$);
                  return TRUE;
              endif
              if SIGNATURE-NAME-CONFLICT(s, $C_i$) then
                  DeleteLink(C, $C_i$);
                  return TRUE;
              endif
          endif
        endfor
        for each direct-superclass $C_k$ of $C_j$ do
          if $C_k \notin$ VISITED then
             VISITED := VISITED $\cup$ {$C_k$};
             QUEUE := QUEUE $\cup$ {$C_k$};
          endif
        endfor
    endwhile
    return FALSE;
end
```

**Algorithm 15: Is-a semantics violation detection algorithm**

Let IS-A-SEMANTICS($C_i$, $C_j$, C) be a function which returns the is-a semantics, either "INCLUSIVE" or "EXCLUSIVE" , between two direct-subclasses $C_i$ and $C_j$ of a class C. Let REQUIRED- IS- A-SEMANTICS($C_i$, $C_j$, C) be a function which first prompts the user to input the required is-a semantics between a direct-subclass $C_i$ of C and a class $C_j$ which is to be made a direct-subclass of C, and then it returns the supplied semantics.

To ensure that an inheritance-link added to make Ci a direct-subclass of C does not violate is-a semantics we use the algorithm :

```
function IS-A-SEMANTICS-VIOLATION(C, $C_i$): BOOLEAN
begin
    for each superclass $C_j$ of $C_i$ do
      if $C_j \in$ Superclasses(C) then
         if the is-a semantics between C and $C_i$ in $C_j$ is exclusive then
            return TRUE;
         endif
         exit; /* this for loop */
      endif
    endfor
     for each direct-subclass $C_k$ of $C_i$ do
         if the required is-a semantics between C and $C_k$ is exclusive then
            if Subclasses(C) $\cap$ Subclasses($C_k$) $\neq \emptyset$ then
               return TRUE;
            endif
         endif
     endfor
     return FALSE;
end
```

The detailed algorithm is:

```
function IS-A-SEMANTICS-VIOLATION(C, $C_i$): BOOLEAN
begin
   S1 := Superclasses(C); QUEUE := $\emptyset$; VISITED := $\emptyset$;
   QUEUE := QUEUE $\cup\{C_i\}$; VISITED := VISITED $\cup\{C_i\}$;
   while QUEUE $\neq \emptyset$ do
       QUEUE := QUEUE - $\{C_j\}$; /* delete $C_j$ from QUEUE */
       for each direct-superclass $C_k$ of $C_j$ do
         if $C_k \notin$ VISITED then
               if $C_k \in$ S1 then
                  for each direct-subclass $C_m$ of $C_k$ do
                      if Cm $\in$ S1 then
                         if IS-A-SEMANTICS($C_m$, $C_j$, $C_k$) = "EXCLUSIVE" then
                             return TRUE;
                         else
                             exit ; /* this while loop */
                         endif
                  endfor
               else
                  VISITED := VISITED $\cup\{C_k\}$;
                  QUEUE := QUEUE $\cup\{C_k\}$;
               endif
         endif
       endfor
   endwhile
```

```
  S2 := Subclasses(C);
  for each direct-subclass $C_j$ of $C_i$ do
    if the required is-a semantics between C and $C_j$ is exclusive then
      if S2 $\cap$ Subclasses($C_j$) $\neq \varnothing$ then
        return TRUE;
      endif
    endif
  endfor
  return FALSE;
end
```

**Algorithm 16: Is-a semantics compatibility detection algorithm**

To ensure that an inheritance-link which is to be added to make C a direct-subclass of Ci will not cause is-a semantics incompatibilities we use the following algorithm before the addition of the link :

```
function IS-A-SEMANTICS-COMPATIBLE(C, $C_i$): BOOLEAN
begin
   for each direct-subclass $C_j$ of $C_i$ do
      /* Determine the required is-a semantics, before the link is added */
      Required-semantics := REQUIRED-IS-A-SEMANTICS($C_j$, C, $C_i$);
       G := Subclasses($C_j$) $\cup$ {$C_j$};
```

```
            for each $C_n \in G$ do
              Perform a superclass breadth-first search starting at $C_n$. The
              search should exclude any path leading to $C_i$;
             if the search intersects Superclasses(C) at $C_k$ then
              if the appropriate is-a semantics at $C_k \neq$ Required-semantics then
                return FALSE;
              endif
             endif
            endfor
        endfor
        return TRUE;
    end
```

# CHAPTER 6

# AUTOMATED TELLER MACHINE (ATM): A CASE STUDY

## 6.1 Problem statement

It is required to design a software to support an ATM network serving a number of banks. The ATM system consists of a number of ATM terminals connected to bank computers via a central computer. Each bank maintains its own computer network.

Each ATM terminal consists of a display screen, a card reader, a cash dispenser, a travellers cheque dispenser, a deposits drawer, a receipt printer, and a keyboard consisting of a cancel key, an enter key, numeric input keys and special input keys used for menu item selection and the selection of a yes or no alternative on a prompt.

The system performs the following financial services for a bank customer: Cash withdrawals in local currency, Funds transfer (in local currency) between Savings and Current accounts, Travellers cheque purchases (in US dollars, Pound Sterlings, and Deuch Marks), Cheque deposits, Statement requests, and Cheque book orders.

The system should:

1. Provide appropriate recordkeeping, error handling, and cryptographic security provisions.
2. Provide the customer with an alternative, whenever possible, if a requests cannot be satisfied.

3. Give the customer the ability to cancel a chosen transaction any time he is prompted for a response.
4. Give a customer an opportunity to verify the amount he entered and change it if necessary before commitment.
5. Be capable of handling multiple transactions for any transaction type other than a cash withdrawal or a travellers cheque purchase transaction.
6. Enable a terminal to continue to function in case it runs out of receipts or in case of receipt ejection error. It should also enable a terminal to provide other financial services in case it runs out of cash or a particular type of travellers cheques.
7. Provide a central computer user interface where a user can issue commands to read and/or modify ATM attributes, (ii) read, delete, or modify the contents of ATM container classes.
8. Provide each bank with a user interface where a user can issue commands to read, delete, or print the contents of ATM class instances of that particular bank.

Access to the system is provided by using an ATM card and a secret four digit personal identification number (PIN). Imprinted on the magnetic stripe of the card is the primary account number (PAN) and the card's serial number. The PAN consists of a unique issuer bank identification number and a unique customers account number. Each card can access one savings and one current account.

Each ATM terminal has a unique identification number. This number accompanies each request from the terminal to the central computer. It is through terminal identity numbers that the central computer identifies ATM terminals sending requests. When the central computer passes a transaction request to a particular bank computer, it sends along with the request the identity number of the ATM terminal at which the request originated. The bank computer in turn sends request results to the central computer accompanied with this terminal identity number.

When an ATM terminal is idle, a prompt to insert an ATM card is displayed. The keyboard and deposit drawer remain inactive until card is inserted. To initiate a transaction a customer inserts his card into the card reader. If the card is unreadable, the customer is informed, and the card is ejected, otherwise the ATM terminal requests the central computer to validate the PAN's bank code. If the bank code is invalid, the customer is informed, and the card is ejected, otherwise the ATM terminal requests the appropriate bank computer, via the central, to verify card validity.

If the validity of the card has expired or if it has been temporarily revoked, the customer is informed and the card is retained, otherwise the customer is prompted to enter his PIN using the keyboard. For each digit entered an X is echoed on the screen. The PIN is encrypted and is sent, along with the PAN, to the central computer and subsequently to the appropriate bank computer for PIN authentication. The maximum number of PIN entry tries and/or PIN entry time-outs is three. If this maximum is exceeded, the customer is informed and the card is retained. If the entered PIN corresponds to the customer's PAN the main menu is displayed. An ATM terminal has a two-level hierarchical menu depicted in Figure 6.1.

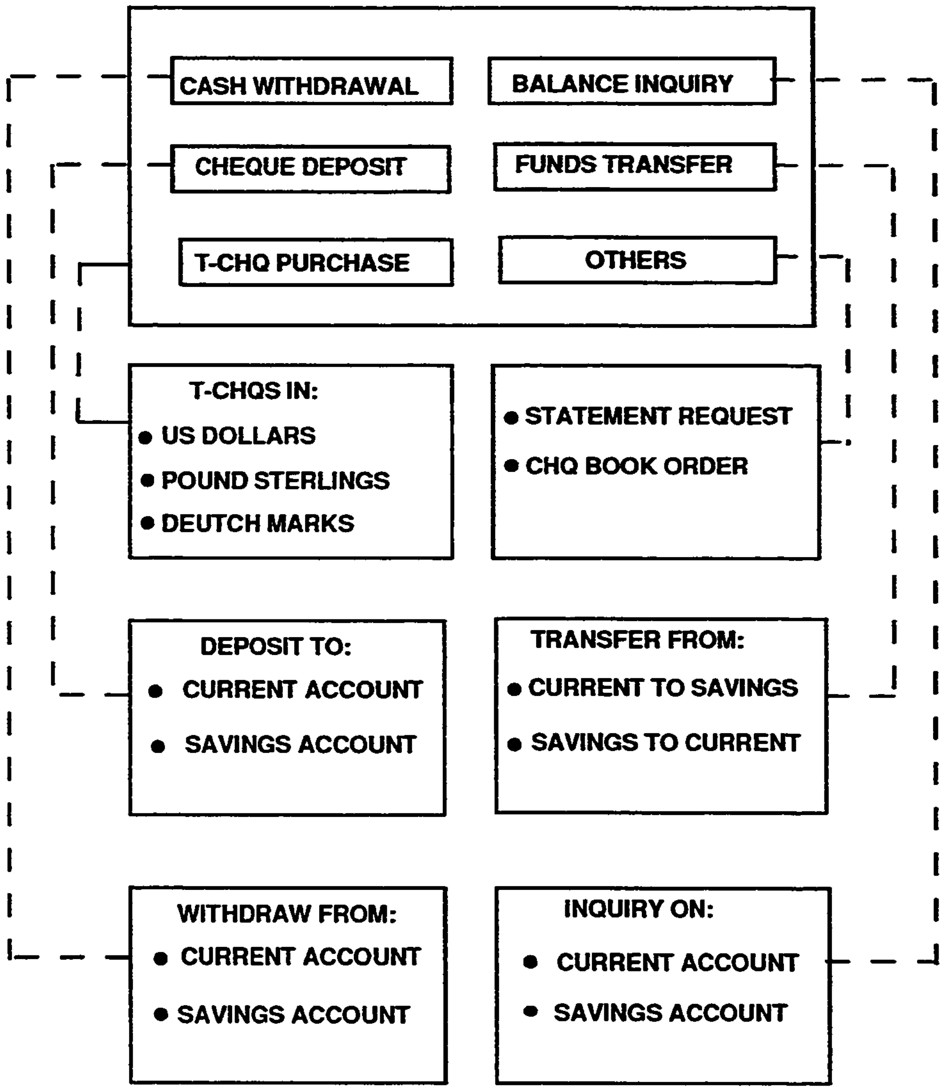


Figure 6.1: ATM-Terminal menu hierarchy

The maximum number of time-outs on menu item choice and/or non-amount entry prompts for any transaction is four. Similarly, the maximum number of time-outs ad/or wrong amount entry for any transaction requiring amount entry is four. In both cases exceeding the maximum causes a card ejection.

The amount entered for a US dollars, or a Deutch Mark travellers cheque purchase, or a cash withdrawal must be in multiples of 100, whereas for a Pound Sterling travellers cheque purchase it must be in multiples of 50. For a funds transfer or a cheque deposit any positive integral amount may be entered. There is a daily cash withdrawal limit as well as a daily travellers cheque purchase limit fixed at a particular local currency limit. The deduction from a customers account for a travellers cheque purchase is in local currency. The central computer maintains current exchange rates for US dollars, Pound Sterlings, and Deutch Marks.

If a cheque deposit transaction is selected, the customer is requested to insert a deposit envelope and to enter the total cheque amount as a truncated integral value. Upon insertion, the PAN, time, date, amount, and ATM terminal identification number are stamped on the deposit envelope.

At the end of a successful cash withdrawal or a travellers cheque purchase transaction the ejected ATM card must be removed before cash or travellers cheques are dispensed. Failure to remove an ejected card, dispensed cash, or dispensed travellers cheques within 2 minutes of an ejection causes the ATM terminal to issue a warning prompt. Another minute delay in doing so results in the retention of the card, cash, or travellers cheques. The ATM keeps a record of such a retention.

A card ejection error, cash dispense error, or a travellers cheque dispense error causes the ATM terminal to issue a redo transaction request.

## 6.2 ATM high-level design

### 6.2.1 The exploratory stage

The following main roles, external interfaces, initial list of subsystems, and initial list of classes were determined in the exploratory stage:

**Main roles:** An ATM network performs financial services for bank customers. To perform these services the system acts in three different roles:

1. Access-right validator.
2. Cryptographic system.
3. Bank financial transaction server:
   3.1 Cash withdrawal server.
   3.2 Travellers cheque purchase server.
   3.3 Cheque deposit server.
   3.4 Funds transfer server.
   3.5 Account-balance inquiry server.
   3.6 Statement request server.
   3.7 Cheque book order request server.

**External Interfaces:**

- ATM-Terminal-User-Interfaces.
- CentralComputerUserInterface.
- BankComputerUserInterfaces.
- ExternalDatabaseInterfaces

**Initial list of subsystems:**

- User-interface subsystem.
- Access right validator subsystem.
- Financial subsystem.
- Communications subsystem.

**Initial list of classes:**

Menu, DisplayScreen, CardReader, CashDispenser, TravellersChequeDispenser, Deposits-Drawer, ReceiptPrinter, ValidationManager, TransactionManager, Log, CommunicationsManager, EncryptionManager, DecryptionManager.

### 6.2.2 The division of the design task into sub-tasks

Since the system to be designed is not complex, there was no need to divide the design task into sub-tasks.

### 6.2.3 The extraction of additional subsystems and classes

The following role-responsibilities were determined:

**Responsibilities as an access-right validator:**

- Verify bank code.
- Verify card validity.
- Authenticate PIN.
- Eject card if the bank code is invalid.
- Retain card if its validity has expired
- Retain card if PIN entry tries exceed 3.
- Provide access to the system.

**Responsibilities as a Cryptographic system:**

- Encrypt PIN.
- Decrypt PIN.
- Transmit the encrypted PIN from an ATM terminal to a bank computer.

**Responsibilities as a Bank financial transaction server:**

**1. Cash withdrawal server.**

- Accept a cash withdrawal request.
- Prompt a customer for the withdrawal account.
- Prompt a customer for the withdrawal amount.
- Ensure that the withdrawal amount is of right multiple.
- Prompt a customer to verify whether the amount he entered is correct.

- Cancel a transaction if selection tries or amount-entry tries exceed the given limit.
- Verify that there is sufficient cash in an ATM terminal to satisfy a request.
- Issue a warning if cash in an ATM terminal gets below some critical value.
- Verify that a customer has not exceeded the daily cash withdrawal limit.
- Verify that the withdrawal amount is within ActualAccntBlnc - MinAccntBlnc.
- Debit from Account balance.
- Modify a customer's daily cash withdrawal amount.
- Dispense cash.
- Detect cash dispense error, if any.
- Know if some or all cash has not been removed after some time limit after being dispensed. Retain this cash.
- Redo a cash withdrawal transaction if there is a card ejection error, or a cash dispense error.
- Issue a receipt for a successful cash withdrawal transaction.
- Record a successful cash withdrawal transaction.
- Inform a customer of any transaction error.

**2. Travellers cheque purchase server:**

- Accept a customer's Travellers cheque purchase request.
- Prompt a customer for debit account.
- Prompt a customer for the type of travellers cheques.
- Prompt a customer for the amount of travellers cheques.
- Ensure that the amount is of right multiple for the selected travellers cheque type.
- Prompt a customer to verify whether the amount he entered is correct

- Verify that there are sufficient travellers cheques of the selected type in an ATM terminal to satisfy a request.
- Convert a travellers cheque amount to its local currency equivalent.
- Prompt a customer to verify whether a travellers cheque purchase request should proceed at the current exchange rate.
- Issue a warning if any travellers cheque amount gets below some critical value.
- Cancel a transaction if selection tries or amount-entry tries exceed the given limit.
- Verify that a customer has not exceeded the daily travellers cheque purchase amount limit.
- Verify that a travellers cheque purchase amount is within ActualAccntBlnc - MinAccntBlnc.
- Debit from Account balance.
- Modify a customers daily travellers cheque purchase amount.
- Dispense travellers cheques.
- Detect travellers cheque dispense error, if any.
- Know if some or all travellers cheques have not been removed after some time limit after being dispensed. Retain these cheques.
- Redo a travellers cheque purchase transaction if there is a travellers cheque dispense error or a card ejection error.
- Issue a receipt for a successful travellers cheque purchase transaction.
- Record a successful travellers cheque purchase transaction.
- Inform a customer of any transaction error.

**3. Cheque deposit server:**

- Accept a cheque deposit request.
- Prompt a customer to insert a deposit envelope.
- Prompt a customer for a deposit account.
- Prompt a customer for a cheque deposit amount.
- Prompt a customer to verify whether the amount he entered is correct.
- Cancel a cheque deposit transaction if selection tries or amount entry tries exceed the given limit.
- Stamp a deposit envelope.
- Record a cheque deposit transaction.
- Issue a receipt for a cheque deposit transaction.
- Inform a customer of any transaction error.

**4. Funds transfer server:**

- Accept a funds transfer request.
- Prompt customer for transfer-from and transfer-to accounts.
- Prompt customer for transfer amount.
- Prompt customer to verify whether the amount he entered is correct.
- Cancel a transaction if selection tries or amount-entry tries exceed the given limit.
- Verify that the amount to be transferred is within ActualAccntBlnc - MinAccntBlnc for the transfer-from account.
- Deduct from transfer-from account.
- Deposit to transfer-to account.
- Record funds transfer transaction.
- Issue a receipt for a funds transfer transaction.
- Inform a customer if there is a transaction error.

**5. Account balance inquiry server**

- Accept an account balance inquiry request.
- Prompt a customer for an account.
- Get account balance.
- Report account balance to a customer.
- Issue a receipt for a balance inquiry transaction.
- Inform a customer of any transaction error.

**6. Statement request server**

- Accept a statement request.
- Record a customer request if a similar request has not been recorded.
- Issue a receipt for a statement request transaction.
- Inform a customer of any transaction error.

**7. Cheque-book order request server**

- Accept a cheque book order request.
- Record a customer request if a similar request has not been recorded.
- Issue a receipt for a cheque book order request transaction.
- Inform a customer of any transaction error.

These role-responsibilities were assigned to classes as shown below. The collaborators of each class were determined. More classes were found by generalisation, specialisation, composition, and decomposition. After evaluation and refinement the class-structure shown in Figure 6.2 and the subsystem collaborations shown in Figure 6.3 were obtained.

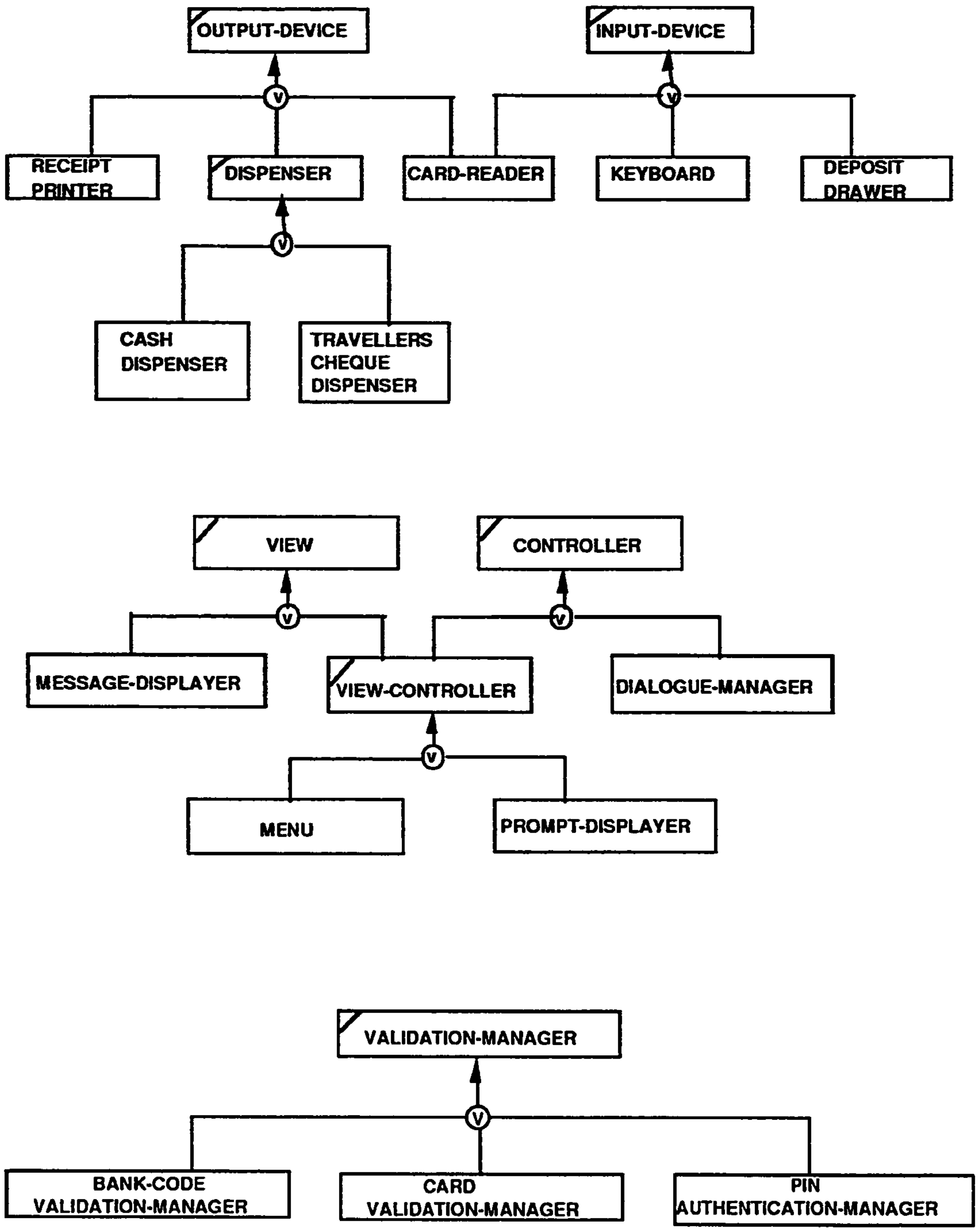
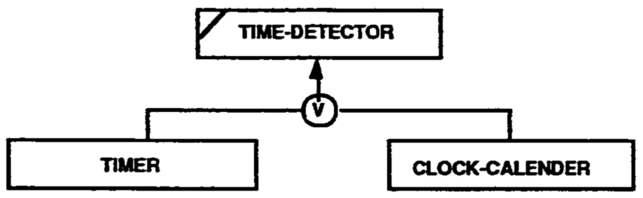
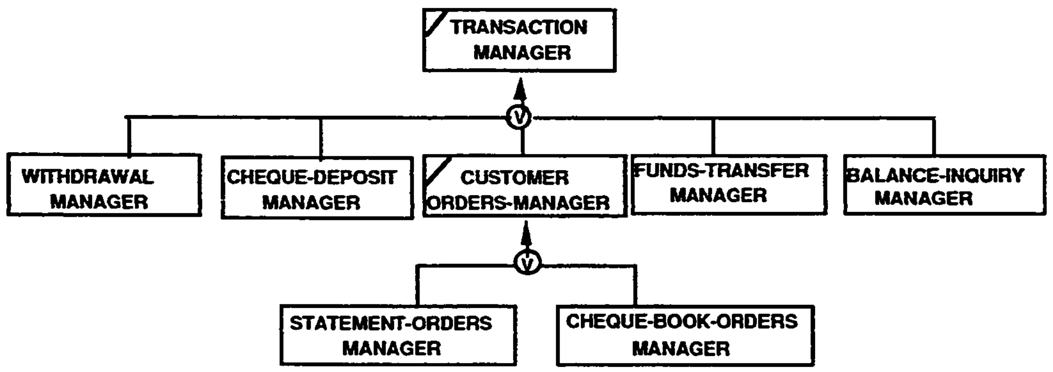


**Figure 6.2: ATM-system class-structure**





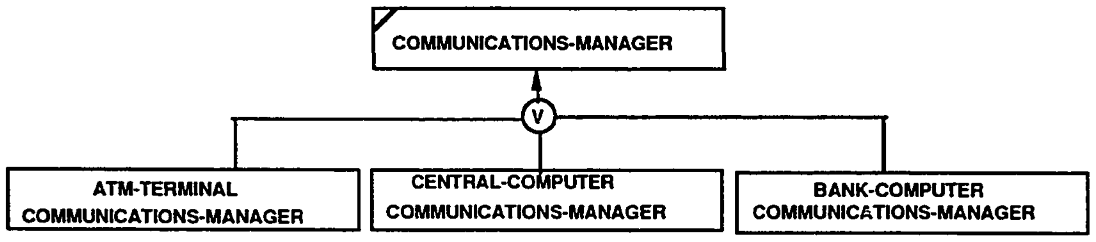

COMMUNICATIONS-MANAGER
V
ATM-TERMINAL COMMUNICATIONS-MANAGER
CENTRAL-COMPUTER COMMUNICATIONS-MANAGER
BANK-COMPUTER COMMUNICATIONS-MANAGER

Figure 6.2 (Continued)

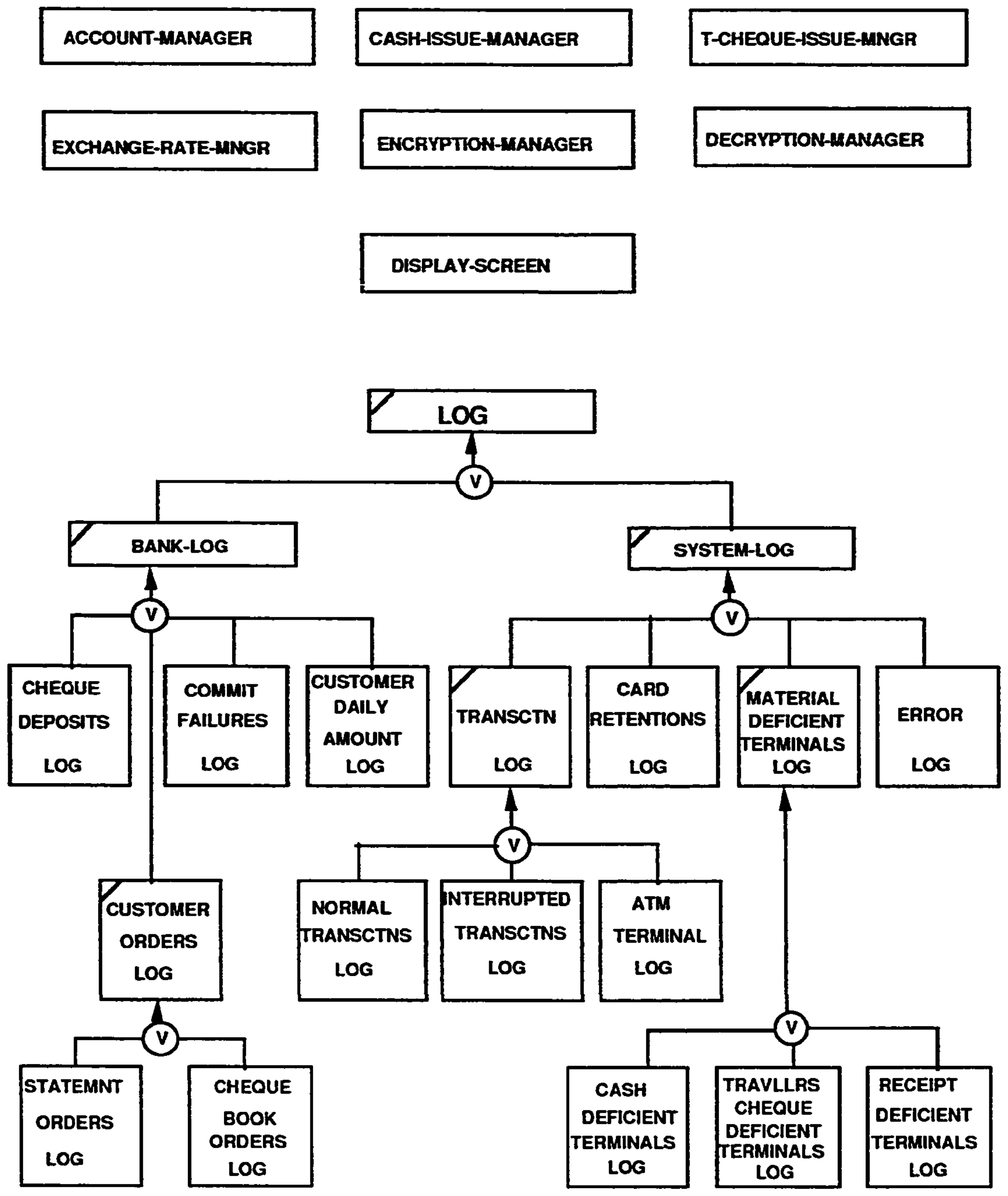


Figure 6.2 (Continued)

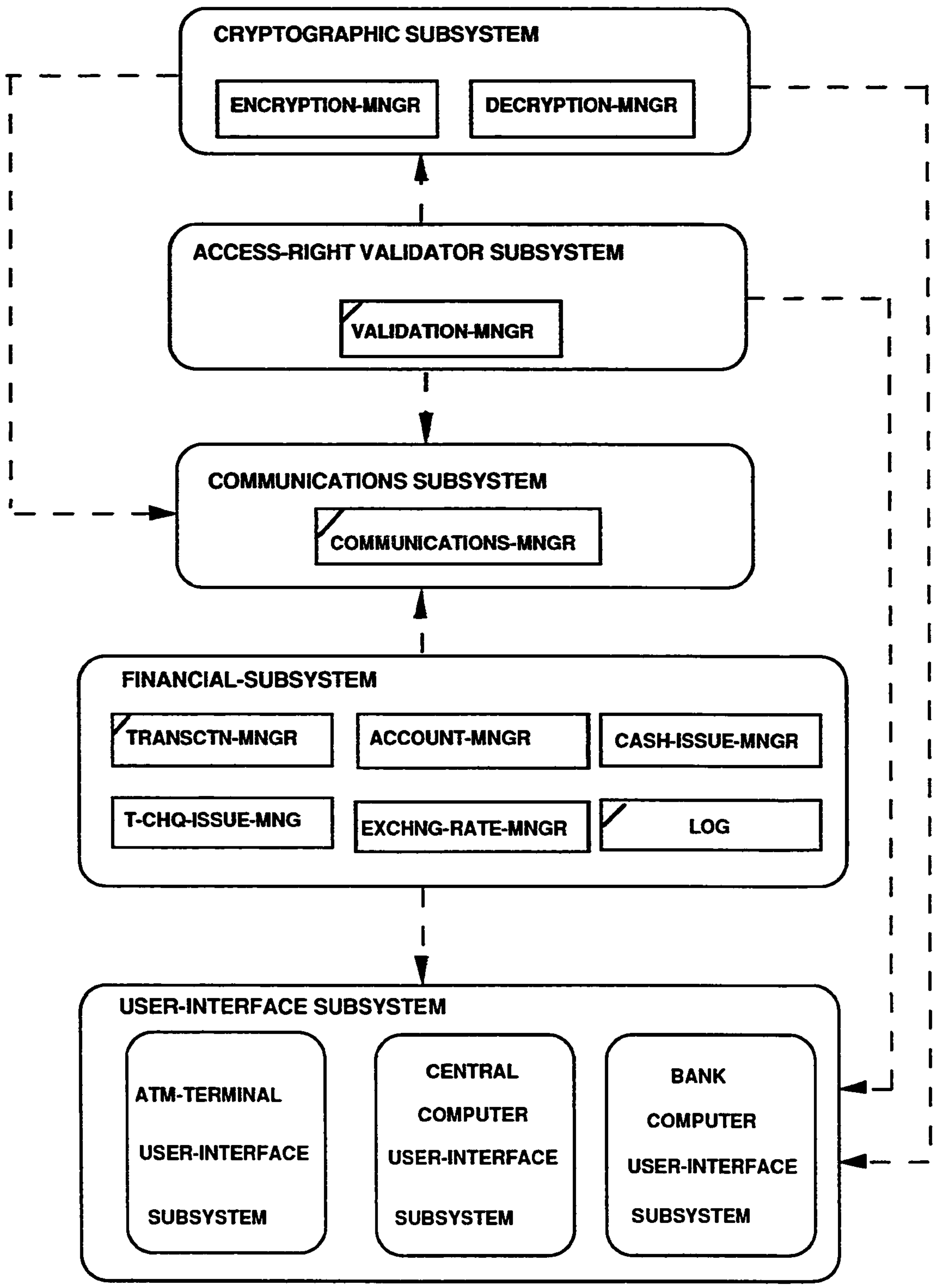


Figure 6.3: ATM-system subsystem-collaborations

**Class-responsibilities and Class-collaborators:**

| Class-responsibilities | Class-collaborators |
|---|---|
| **VIEW** | |
| ● Initiates displays | DisplayScreen. |
| **CONTROLLER** | |
| ● Gets user commands, interprets them, and dispatches them to an appropriate class. | Keyboard. |
| **MENU** | |
| ● Displays itself. | DisplayScreen. |
| ● Gets a menu command from a user. | Keyboard. |
| ● Executes a menu command. | |
| ● Dispatches results to DialogueManager. | DialogueManager. |
| ● Knows user response time. | Timer. |
| **PROMPT DISPLAYER** | |
| ● Displays prompts. | DisplayScreen. |
| ● Displays numeric values. | DisplayScreen. |
| ● Gets user response. | Keyboard. |
| ● Dispatches user response to Dialogue Manager. | DialogueManager. |
| ● Knows user response time. | Timer. |
| **MESSAGE DISPLAYER** | |
| ● Displays messages. | DisplayScreen. |
| ● Displays numeric values. | DisplayScreen. |

**DISPLAY SCREEN**

- Displays messages.
- Displays prompts.
- Displays numeric values.
- Displays menus.

**ATM TERMINAL DIALOGUE MANAGER**

| | |
|---|---|
| ● Check card insertion. | CardReader. |
| ● Initiates a card read. | CardReader. |
| ● Gets the results of a card read. | CardReader. |
| ● Initiates the display of prompts. | PromptDisplayer. |
| ● Initiates the display of messages. | MessageDisplayer. |
| ● Gets user response. | Menu, PromptDisplayer, Keyboard, InputDevice. |
| ● Initiates access-right validation. | ValidationManager. |
| ● Gets the results of access-right validation. | ValidationManager. |
| ● Initiates PIN authentication. | EncryptionManager. |
| ● Gets the result of PIN authentication. | PIN-AuthenticationManager. |
| ● Counts PIN-entry tries. | PromptDisplayer, Keyboard. |
| ● Initiates the display of the main menu. | Menu. |
| ● Counts selection tries | PromptDisplayer, Menu, Keyboard. |
| ● Check deposit envelope insertion. | DepositDrawer. |
| ● Counts amount-entry tries. | PromptDisplayer, Keyboard. |
| ● Initiates the stamping of the deposit envelope. | DepositDrawer. |

| | |
|---|---|
| • Ensures that the entered amount is of right multiple for the requested transaction. | |
| • Verifies that there is sufficient cash in an ATM terminal to satisfy a cash withdrawal request. | CashIssueManager. |
| • Verifies that there are sufficient travellers cheques of the selected type in an ATM terminal to satisfy a travellers cheque purchase request. | TravellersChequeIssueManager. |
| • Converts a travellers cheque amount to local currency equivalent. | ExchangeRateManager. |
| • Requests the time and the date a transaction is initiated. | Timer, ClockCalendar. |
| • Dispatches transaction requests. | TransactionManager. |
| • Initiates a card ejection. | CardReader. |
| • Initiates a receipt print and ejection. | ReceiptPrinter. |
| • Requests the recording of a successful cash withdrawal, travellers cheque purchase, or a funds transfer transaction. | TransactionLog. |
| • Initiates card retention. | CardReader. |
| • Initiates cash retention. | CashDispenser. |
| • Initiates travellers cheque retention. | TravellersChequeDispenser. |
| • Requests the recording of ATM terminal errors. | ErrorLog. |
| • Initiates a redo for a cash withdrawal or a travellers cheque purchase transaction. | DepositManager. |

| | |
|---|---|
| ● Requests the recording of a card retention including the reason for the retention. | CardRetentionsLog. |
| ● Requests the recording of transactions which result in cash or travellers cheque retention. | ATM-TerminalLog. |

**CENTRAL COMPUTER DIALOGUE MANAGER**

| | |
|---|---|
| ● Initiates the display of prompts. | CentralComputerPromptDisplayer. |
| ● Initiates the display of messages. | CentralComputerMessageDisplayer. |
| ● Gets user response. | CentralComputerMenu, CentralComputerPromptDisplayer, CentalComputerKeyboard, CentralComputerInputDevice. |
| ● Initiates the display of the main menu. | CentralComputerMainMenu. |
| ● Initiates a read, delete or print operation on InterruptedTransactionsLog. | InterruptedTransactionsLog, CentralComputerPrinter. |
| ● Initiates a read, delete, or print operation on CardRetentionsLog. | CardRetentionsLog, CentralComputerPrinter. |
| ● Initiates a read, delete, or print operation MaterialDeficientTerminalsLog. | MaterialDeficientTerminalsLog. |
| ● Initiates a read, delete, or print operation on an ATM-TerminalLog. | ATM-TerminalLog. |
| ● Initiates a read or modify operation on an ATM-Terminal attribute. | |
| ● Initiates an add, delete, or modify bank code operation. | BankCodeValidationManager. |

**BANK COMPUTER DIALOGUE MANAGER**

| | |
|---|---|
| ● Initiates the display of prompts. | BankComputerPromptDisplayer. |
| ● Initiates the display of messages. | BankComputerMessageDisplayer. |
| ● Gets user response. | BankComputerMenu, BankComputerPromptDisplayer, BankComputerKeyboard, BankComputerInputDevice. |
| ● Initiates the display of the main menu. | BankComputerMainMenu. |
| ● Initiates a restricted read or print operation on TransactionLog. | TransactionLog. |
| ● Initiates a restricted read or print operation on InterruptedTransactionsLog. | InterruptedTransactionsLog. |
| ● Initiates a restricted read or print operation on CardRetentionsLog. | CardRetentionsLog. |
| ● Initiates a read, delete, or print operation on the banks ATM Logs. | |
| ● Initiates a read or modify operation on the banks ATM classes attributes. | |
| ● Initiates a delete, add, or change PIN operation. | PIN-AuthenticationManager. |
| ● Initiates a temporary revocation of card validity. | ValidationManager. |

**INPUT DEVICE**

● Gets user input.

**KEYBOARD**

- Gets keypress events.
- Sends keypress events to Menu.
- Sends keypress events to PromptDisplayer.

**DEPOSIT DRAWER**

- Accepts an open drawer command from the DialogueManager.
- Detects the insertion of a deposit envelope.
- Stamps the deposit envelope.

**CARD READER**

- Detects a card insertion.
- Reads a card.
- Ejects a card.
- Detects a card ejection error.
- Detects a card removal.
- Retains a card.
- Detects a card retention error.

**OUTPUT DEVICE**

- Gives output.

**CASH DISPENSER**

- Dispenses cash.
- Detects cash dispense error.
- Detects cash removal.
- Retains cash.
- Detects cash retention error.

**TRAVELLERS CHEQUE DISPENSER**

| | |
|---|---|
| • Dispenses travellers cheques. | |
| • Detects travellers cheque dispense error. | |
| • Detects travellers cheque removal. | |
| • Retains travellers cheques. | |
| • Detects travellers cheque retention error. | |

**RECEIPT PRINTER**

| | |
|---|---|
| • Prints receipt for a transaction. | |
| • Ejects receipts. | |
| • Detects Printer error. | |
| • Detects receipt ejection error. | |
| • Issues a warning when the number of receipts gets below a critical value. | MaterialDeficientLog. |

**CASH ISSUE MANAGER**

| | |
|---|---|
| • Issues cash. | CashDispenser. |
| • Issues a warning when cash-in-hand gets below critical value. | CashDeficientLog. |

**TRAVELLERS CHEQUE ISSUE MANAGER**

| | |
|---|---|
| • Issues travellers cheques. | TravellersChequeDispenser. |
| • Issues a warning when amount-in-hand gets below critical value. | TravellersChequeDeficientLog. |

**EXCHANGE RATE MANAGER**

• Maintains current exchange rates for US dollars, PoundSterlings, and Deutch Marks.

**VALIDATION MANAGER**

• Validates the access rights of a user.

**BANK CODE VALIDATION MANAGER**

- Validates a PAN's bank code.

**CARD VALIDATION MANAGER**

- Verifies the validity of an ATM card.

**ENCRYPTION MANAGER**

- Encrypts PIN.

**DECRYPTION MANAGER**

- Decrypts an encrypted PIN.

**PIN AUTHENTICATION MANAGER**

- Authenticates PIN.

**TIME DETECTOR**

- Detects one second events.

**TIMER**

- Times user response.

**CLOCK CALENDAR**

- Supplies the current date and time.

**TRANSACTION MANAGER**

- Executes a financial transaction for a particular bank.
- Sends results to DialogueManager.

**WITHDRAWAL MANAGER**

| | |
|---|---|
| • Executes a cash withdrawal transaction. | AccountManager, CommitFailures-Log, DailyAmountLog. |
| • Executes a travellers cheque purchase | AccountManager, CommitFailures-Log, DailyAmountLog. |

**DEPOSIT MANAGER**

| | |
|---|---|
| • Executes a cheque deposit transaction. | ChequeDepositsLog. |
| • Redo a cash withdrawal transaction. | AccountManager, DailyAmountLog. |
| • Redo a travellers cheque purchase. | AccountManager, DailyAmountLog. |

**FUNDS TRANSFER MANAGER**

| | |
|---|---|
| • Executes a funds transfer transaction. | AccountManager, CommitFailures-Log. |

**BALANCE INQUIRY MANAGER**

| | |
|---|---|
| • Executes a balance inquiry. | AccountManager, CommitFailures-Log. |

**STATEMENT ORDERS MANAGER**

| | |
|---|---|
| • Executes a statement order request. | StatementOrdersLog. |

**CHEQUE BOOK ORDERS MANAGER**

| | |
|---|---|
| • Executes a cheque book order request. | ChequeBookOrdersLog. |

**ACCOUNT MANAGER**

- Accesses bank accounts.
- Modifies bank accounts.

**BANK LOG**

- Records transactions or there aspects for a particular bank.

**CHEQUE DEPOSITS LOG**

- Records cheque deposits for a particular bank.

**COMMIT FAILURES LOG**

- Records withdrawal commit failures on a given account for a particular bank for those withdrawal transactions in which the stage of accessing the account was successful.
- Records deposit commit failures on a given account for a particular bank.

**DAILY AMOUNT LOG**

- Records the total daily cash withdrawals for each customer for a particular bank.
- Records the total daily travellers cheque purchases, in local currency, of each customer for a particular bank.

**STATEMENT ORDERS LOG**

- Records customer statement orders for a particular bank.

**CHEQUE BOOK ORDERS LOG**

- Records customer cheque book orders for a particular bank.

**SYSTEM LOG**

- Records errors and material deficiencies of the ATM system.
- Records transactions for all banks.

**NORMAL TRANSACTIONS LOG**

- Records all daily cash withdrawals, travellers cheque purchases, and fund transfers for customers.

**INTERRUPTED TRANSACTIONS LOG**

- Records cash withdrawals, travellers cheque purchases, and fund transfers which cannot be completed due to communication breakdown between the Central Computer and one or more ATM terminals and which cannot be redone due to transmission error.

**ATM TERMINAL LOG**

- Records all successful cash withdrawals, travellers cheque purchases, or fund transfers on a particular ATM terminal which cannot be recorded on the TransactionLog due to transmission error.
- Records cash withdrawal or travellers cheque purchase transactions on a particular ATM terminal in which a customer failed to remove cash or travellers cheques after a given time limit.
- Records withdrawal or travellers cheque purchase transactions on a particular ATM terminal which are not completed due to either card ejection error, cash dispense error, or cheque dispense error and which cannot be redone due to transmission error.

**CARD RETENTIONS LOG**

- Records card retentions in all ATM terminals.

**CASH DEFICIENT TERMINALS LOG**

- Records ATM terminal identity numbers of terminals with deficient cash.

**TRAVELLERS CHEQUE DEFICIENT TERMINALS LOG**

- Records ATM terminal identity numbers of terminals with deficient travellers cheques of any kind.

**RECEIPT DEFICIENT TERMINALS LOG**

- Records ATM terminal identity numbers of terminals with deficient receipts.

**ERROR LOG**

- Records ATM terminal identity numbers of terminals with either card ejection error, cash dispense error, travellers cheque dispense error, receipt ejection error, card retention error, cash retention error, or travellers cheque retention error.

**COMMUNICATIONS MANAGER**

- Manages communications.
- Detects transmission errors.

## 6.2.4. The construction of class interfaces

The following class signatures and attributes were determined:

### Class signatures and attributes:

**WITHDRAWAL MANAGER**

MaxDailyWithdrawalAmount: Int

MaxDailyChequePurchaseAmount: Real

CustomerInCommitFailuresLog: Boolean

CustomerInDailyAmountLog: Boolean

WithdrawCash(PAN: Int; ATM-Terminal-id: Int; Accnt: String)

GetMaxDailyWithdrawalAmount( ): Int

GetMaxDailyChequePurchaseAmount( ): Real

ModifyMaxDailyWithdrawalAmount(NewMax: Int)

ModifyMaxDailyChequePurchaseAmount(NewMax: Real)

**ACCOUNT MANAGER**

MinSavingsBalance: Real

MinCurrentBalance: Real

AccessBalance(PAN: Int; Accnt: String): Boolean

GetBalance(PAN: Int; Accnt: String): Real

AddToBalance(PAN: Int; Accnt: String; Amnt: Real)

DeductFromBalance(PAN: Int; Accnt: String; Amnt: Real)

GetMinSavingsBalance( ): Real

GetMinCurrentBalance( ): Real

ModifyMinSavingsBalance(NewBlnc: Real)

ModifyMinCurrentBalance(NewBlnc: Real)

**DEPOSIT MANAGER**

DepositCheque(PAN: Int; ATM-Terminal-id: Int; Amnt: Int; Accnt: String)

RedoCashWithdrawal(PAN: Int; ATM-Terminal-id; Amnt: int; Accnt: String)

RedoTravellersChequePurchase(PAN: Int; ATM-Terminal-id: Int; Amnt: Real; Accnt: String)

**FUNDS TRANSFER MANAGER**

CustomerInCommitFailuresLog: Boolean

TransferFunds(PAN: Int; ATM-Terminal-id: Int; TransferCode: Int; Amnt: Real)

**BALANCE INQUIRY MANAGER**

Inquire-on-balance(PAN: Int; ATM-Terminal-id: Int; Accnt: String)

**STATEMENT ORDERS MANAGER**

OrderStatement(PAN: Int; ATM-Terminal-id: Int; Accnt: String)

**CHEQUE BOOK ORDERS MANAGER**

OrderChequeBook(PAN: Int; ATM-Terminal-id: Int)

**EXCHANGE RATE MANAGER**

US-DollarExchangeRate: Real

PoundSterlingExchangeRate: Real

DeutchMarkExchangeRate: Real

GetExchangeRate(CurrencyType: String): Real

ModifyExchangeRate(CurrencyType: String; NewRate: Real)

**TRAVELLERS CHEQUE ISSUE MANAGER**

USD-Amnt-in-hand: Int

PoundSterlingAmnt-in-hand: Int

DeutchMarkAmnt-in-hand: Int

USD-CriticalAmnt: Int

PoundSterlingCriticalAmnt: Int

DeutchMarkCriticalAmnt: Int

USD-Amnt-is-critical: Boolean

PoundSterlingAmnt-is-critical: Boolean

DeutchMarkAmnt-is-Critical: Boolean

GetChequeAmntInHand(CurrencyType: String): Int

GetCriticalChequeAmnt(CurrencyType: String): Int

SubtractFromChequeAmntInHand(CurrencyType: String; Amnt: Int)

ModifyChequeAmntInHand(CurrencyType: String; NewAmnt: Int)

SetCriticalAmntFlag(CurrencyType: String)

ModifyCriticalChequeAmnt(CurrencyType: String; NewAmnt: Int)

IssueCheques(CurrencyType: String; Amnt: Int)

**TRAVELLERS CHEQUE DISPENSER**

DispenseCheques(CurrencyType: String; Amnt: Int): Boolean

RetainCheques( ): Boolean

**CASH ISSUE MANAGER**

CashInHand: Int

CriticalCashAmnt: Int

AmntIsCritical: Boolean

GetCashInHand( ): Int

GetCriticalCashAmnt( ): Int

ModifyCashInHand(NewAmnt: Int)

ModifyCriticalCashAmnt(NewAmnt: Int)

SubtractFromCashInHand(Amnt: Int)

SetCriticalAmntFlag( )

IssueCash(Amnt: Int)

**CASH DISPENSER**

DispenseCash(Amnt: Int): Boolean

RetainCash( ): Boolean

CashRemoved( ): Boolean

**LOG**

ReadLog( )

DeleteLogContents( )

PrintLogContents( )

**BANK LOG**

CustomerIsRecordedInLog(PAN: Int): Boolean

RecordCustomerInLog(PAN: Int)

DeleteCustomerFromLog(PAN: Int)

**CHEQUE DEPOSITS LOG**

RecordChequeDepositTransaction(PAN:Int;ATM-Terminal-id:Int;Amnt:Int)

**COMMIT FAILURES LOG**

AddToCustomerWithdrawalFailures(PAN: Int; Amnt: Real; Accnt: String)

AddToCustomerDepositFailures(PAN: Int; Amnt: Real; Accnt: String)

GetCustomerWithdrawalFailures(PAN: Int; Accnt: String): Real

GetCustomerDepositFailures(PAN : Int; Accnt: String): Real

DeleteCustomerWithdrawalFailures(PAN: Int; Accnt: String)

DeleteCustomerDepositFailures(PAN: Int; Accnt: String)

**DAILY AMOUNT LOG**

AddToCustomerDailyCashWithdrawal(PAN: Int; Amnt: Int)

AddToCustomerDailyTravellersChequeAmnt(PAN: Int; Amnt: Real)

SubtractFromCustomerDailyCashWithdrawal(PAN: Int; Amnt: Int)

SubtractFromCustomerDailyTravellersChequeAmnt(PAN: Int; Amnt: Real)

GetCustomerDailyCashWithdrawal(PAN: Int): Int

GetCustomerDailyTravellersChequeAmnt(PAN: Int): Real

**TRANSACTION LOG**

RecordCustomerWithdrawalTransaction(PAN: Int; Amnt: Int; Accnt: String;ATM-Terminal-id: Int; Time: Real; Day: Date)

RecordCustomerTravellersChequeTransaction(PAN : Int ; Amnt : Int ; CurrencyType: String; ExchangeRate: Real; Accnt: String; ATM-Terminal-id: Int; Time: Real; Day: Date)

RecordCustomerFundsTransferTransaction(PAN: Int; Amnt: Int; TransferCode: Int; ATM-Terminal-id: Int; Time: Real; Day: Date)

ReadBankTransactions(BankCode: Int)

PrintBankTransactions(BankCode: Int)

DeleteBankTransactions(BankCode: Int)

**CARD RETENTIONS LOG**

RecordCardRetention(PAN: Int; ATM-Terminal-id: Int; ReasonForRetention: String; Time: Real; Day: Date)

Read-ATM-TerminalCardRetentions(ATM-Terminal-id: Int)

ReadBankCardRetentions(BankCode: Int)

PrintATM-TerminalCardRetentions(BankCode: Int)

PrintBankCardRetentions(BankCode: Int)

DeleteATM-TerminalCardRetentions(ATM-Terminal-id: Int)

DeleteBankCardRetentions(BankCode: Int)

**ERROR LOG**

RecordATM-TerminalError(ATM-Terminal-id: Int; ErrorType: String; Time: Real; Day: Date)

ReadATM-TerminalError(ATM-Terminal-id: Int)

PrintATM-TerminalError(ATM-Terminal-id: Int)

DeleteATM-TerminalError(ATM-Terminal-id: Int; ErrorType: String)

**CASH DEFICIENT TERMINALS LOG**

RecordCashDeficiency(ATM-Terminal-id: Int; Time: Real; Day: Date)

DeleteCashDeficiency(ATM-Terminal-id: Int)

**RECEIPT DEFICIENT TERMINALS LOG**

RecordReceiptDeficiency(ATM-Terminal-id: Int; Time: Real; Day: Date)

DeleteReceiptDeficiency(ATM-Terminal-id: Int)

**TRAVELLERS CHEQUE DEFICIENT TERMINALS LOG**

RecordTravellersChequeDeficiency(ATM-Terminal-id: Int; CurrencyType: String; Time: Real; Day: Date)

DeleteTravellersChequeDeficiency(ATM-Terminal-id: Int)

**ENCRYPTION MANAGER**

Encrypt(PIN: Int; PAN: Int; ATM-Terminal-id: Int)

**DECRYPTION MANAGER**

Decrypt(EncyptedPIN: Int; PAN: Int; ATM-Terminal-id: Int)

**PIN AUTHENTICATION MANAGER**

Authenticate(PIN: Int; PAN: Int; ATM-Terminal-id: Int)

AddToPIN-Table(PAN: Int)

DeleteFromPIN-Table(PAN: Int)

ChangePIN(NewPIN: Int; PAN: Int)

**BANK CODE VALIDATION MANAGER**

ValidateBankCode(PAN: Int; CardSerialNumber: Int; ATM-Terminal-id: Int)

AddBankCode(BankCode: Int): Boolean

DeleteBankCode(BankCode: Int)

ChangeBankCode(OldCode: Int; NewCode: Int): Boolean

**CARD VALIDATION MANAGER**

ValidateCard(PAN: Int; CardSerialNumber: Int; ATM-Terminal-id: Int)

RevokeCardValidity(CardSerialNumber: Int)

RevokeCardValidityTemporarily(CardSerialNumber: Int)

MakeCardValid(PAN: Int; CardSerialNumber: Int)

**CARD READER**

ReadCard( )

CardIsInserted( ): Boolean

EjectCard( ): Boolean

CardRemoved( ): Boolean

RetainCard( ): Boolean

**RECEIPT PRINTER**

Receipts-in-hand: Int

ReceiptCriticalAmnt: Int

ReceiptAmntIsCritical: Boolean

PrinterOutOfReceipts: Boolean

PrintReceipt(PAN: Int; TrnsctnType: String; Amnt: Int; Blnc: Real; ATM-Terminal-id: Int; Time: Real; Day: Date)

EjectReceipt( ): Boolean

GetReceiptsInHand(ATM-Terminal-id: Int): Int

ModifyReceiptsInHand(ATM-Terminal-id: Int; NewQuantity: Int)

SetCriticalReceiptAmntFlag( )

SetReceiptAmntFlag( )

**KEYBOARD**

GetKeyPressEvent( ): KeyPressEvent

**DISPLAY SCREEN**

DisplayText(Text: String; Coordinate: Point)

DisplayNumbre(Number: Real; Coordinate: Point)

**MESSAGE DISPLAYER**

DisplayMessage(Text: String)

**PROMPT DISPLAYER**

PutUpPromptWindow(Text: String; WindowFrameCode: Int)

**MENU**

GetMenuCommand( ): MenuCommandEvent

ExecuteMenuCommand(CommandNumber: Int)

DrawMenu(Menuframe: Int)

**DEPOSIT DRAWER**

OpenDepositDrawer( )

DepositEnvelopeIsInserted( ): Boolean

StampEnvelope(PAN: Int; ATM-Terminal-id: Int; Time: Real; Day: Date; Amnt: Int)

**TIMER**

Second: Int

TimerOn: Boolean

StartTimer( )

StopTimer( )

ElapsedTime( ): Int

**CLOCK CALENDAR**

Day: IntRange 1 .. 31

Month: IntRange 1 .. 12

Year: IntRange 1992 .. 3000

Hour: IntRange 0 .. 24

Minute: IntRange 0 .. 60

Second: IntRange 0 .. 60

GetDate( ): Date

GetTime( ): Real

**DIALOGUE MANAGER**

ATM-Terminal-id: Int

SelectionTries: Int

AmntEntryTries: Int

PIN-EntryTries: Int

TransactedAmnt: Int

TransactionCode: Int

CheckCardInsertion( )

CardReadResult(CardIsReadable: Boolean; PAN: Int; CardSerialNumber: Int)

BankCodeAuthenticationResult(BankCodeIsAuthentic: Boolean;ATM-Terminal-id: Int)

CardValidationResult(CardIsValid: Boolean; ATM-Terminal-id: Int)

PIN-EntryResult(PIN: Int)

PIN-AuthenticationResult(PIN-Is-Authentic: Boolean; ATM-Terminal-id: Int)

MenuEventHandler(MenuEventCode: Int)

PromptHandler(EventCode: Int)

AmntEntryHandler(Amnt: Int)

TerminalErrorHandler(ErrorCode: Int)

DispatchTransaction(TransactionCode: Int)

Date(Day: Int; Month: Int; Year: Int)

Time(Hour: Int; Min: Int; Sec: Int)

CashWithdrawalResult(CompletionStatus: String; Blnc: Real; ATM-Terminal-id: Int)

T-ChqPurchaseResult(CompletionResult: String; Blnc: Real; ATM-Terminal-id: Int)

FundsTransferResult(CompletionStatus: String; Blnc: Real; ATM-Terminal-id: Int)

ChequeDepositResult(CompletionStatus: String; Blnc: Real; ATM-Terminal-id: Int)

BalanceInquiryResult(CompletionStatus: String; Blnc: Real; ATM-Terminal-id: Int)

StatementOrderResult(CompletionStatus: String; Blnc: Real; ATM-Terminal-id: Int)

ChequeBookOrderResult(CompletionStatus: String; Blnc: Real;ATM-Terminal-id:Int)

### 6.2.5. The construction of the dynamic model

Some class statecharts in the dynamic model we constructed are shown in Figures 6.4, 6.5, 6.6, 6.7, and 6.8.

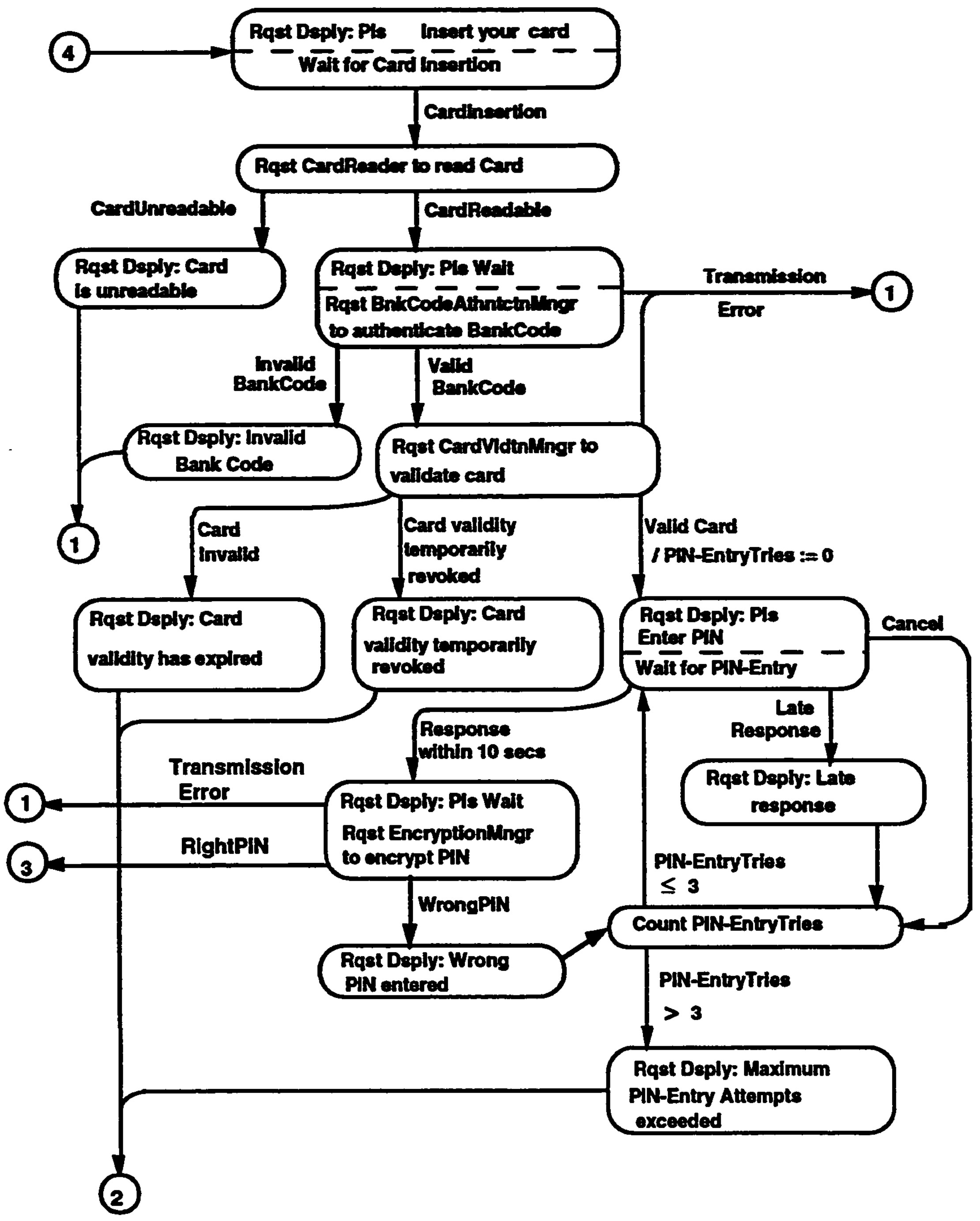
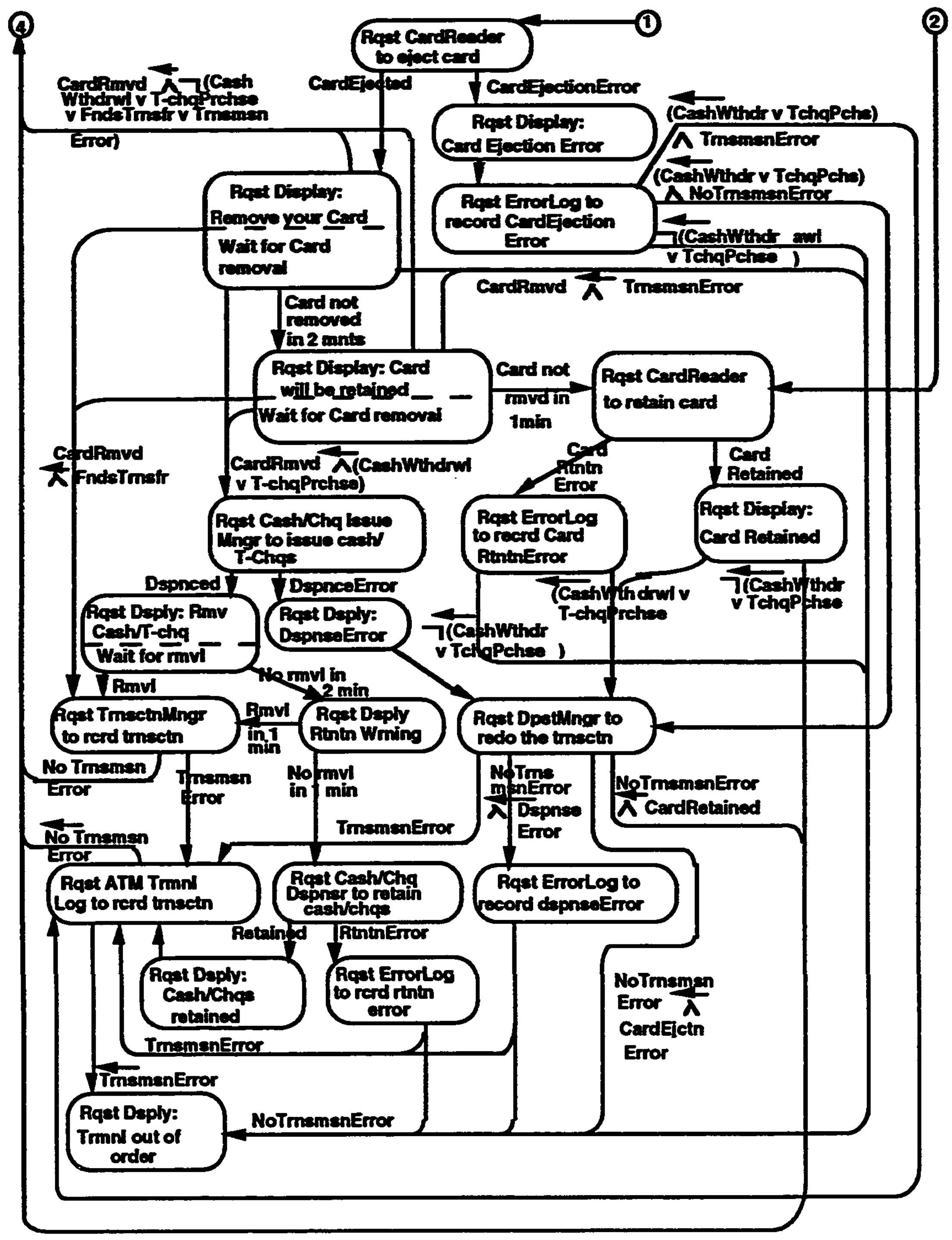

4
Rqst Dsply: Pls Insert your card
Wait for Card Insertion
CardInsertion
Rqst CardReader to read Card
CardUnreadable
CardReadable
Rqst Dsply: Card is unreadable
Rqst Dsply: Pls Wait
Rqst BnkCodeAthntctnMngr to authenticate BankCode
Transmission Error
1
Invalid BankCode
Valid BankCode
Rqst Dsply: Invalid Bank Code
Rqst CardVldtnMngr to validate card
1
Card Invalid
Card validity temporarily revoked
Valid Card / PIN-EntryTries := 0
Rqst Dsply: Card validity has expired
Rqst Dsply: Card validity temporarily revoked
Rqst Dsply: Pls Enter PIN
Wait for PIN-Entry
Cancel
Late Response
Response within 10 secs
Transmission Error
1
Rqst Dsply: Pls Wait
Rqst EncryptionMngr to encrypt PIN
Rqst Dsply: Late response
RightPIN
3
PIN-EntryTries ≤ 3
WrongPIN
Count PIN-EntryTries
Rqst Dsply: Wrong PIN entered
PIN-EntryTries > 3
Rqst Dsply: Maximum PIN-Entry Attempts exceeded
2

Figure 6.4: DialogueManager Statechart



4
1
2
Rqst CardReader to eject card
CardEjected
CardEjectionError
CardRmvd ∧ ¬(Cash Wthdrwl v T-chqPrchse v FndsTrnsfr v Trnsmsn Error)
Rqst Display: Card Ejection Error
(CashWthdr v TchqPchs) ∧ TrnsmsnError
(CashWthdr v TchqPchs) ∧ NoTrnsmsnError
Rqst Display: Remove your Card Wait for Card removal
Rqst ErrorLog to record CardEjection Error
¬(CashWthdr awl v TchqPchse)
CardRmvd ∧ TrnsmsnError
Card not removed in 2 mnts
Rqst Display: Card will be retained Wait for Card removal
Card not rmvd in 1min
Rqst CardReader to retain card
CardRmvd ∧ FndsTrnsfr
CardRmvd ∧(CashWthdrwl v T-chqPrchse)
Card Rtntn Error
Card Retained
Rqst Cash/Chq Issue Mngr to issue cash/ T-Chqs
Rqst ErrorLog to recrd Card RtntnError
Rqst Display: Card Retained
Dspnced
DspnceError
(CashWthdrwl v T-chqPrchse)
¬(CashWthdr v TchqPchse)
Rqst Dsply: Rmv Cash/T-chq Wait for rmvl
Rqst Dsply: DspnseError
¬(CashWthdr v TchqPchse)
No rmvl in 2 min
Rmvl
Rqst TrnsctnMngr to rcrd trnsctn
Rmvl in 1 min
Rqst Dsply Rtntn Wrning
Rqst DpstMngr to redo the trnsctn
No Trnsmsn Error
Trnsmsn Error
No rmvl in 1 min
NoTrns msnError ∧ Dspnse Error
NoTrnsmsnError ∧ CardRetained
TrnsmsnError
No Trnsmsn Error
Rqst ATM Trmnl Log to rcrd trnsctn
Rqst Cash/Chq Dspnsr to retain cash/chqs
Rqst ErrorLog to record dspnseError
Retained
RtntnError
Rqst Dsply: Cash/Chqs retained
Rqst ErrorLog to rcrd rtntn error
NoTrnsmsn Error ∧ CardEjctn Error
TrnsmsnError
TrnsmsnError
Rqst Dsply: Trmnl out of order
NoTrnsmsnError

Figure 6.4 (Continued)

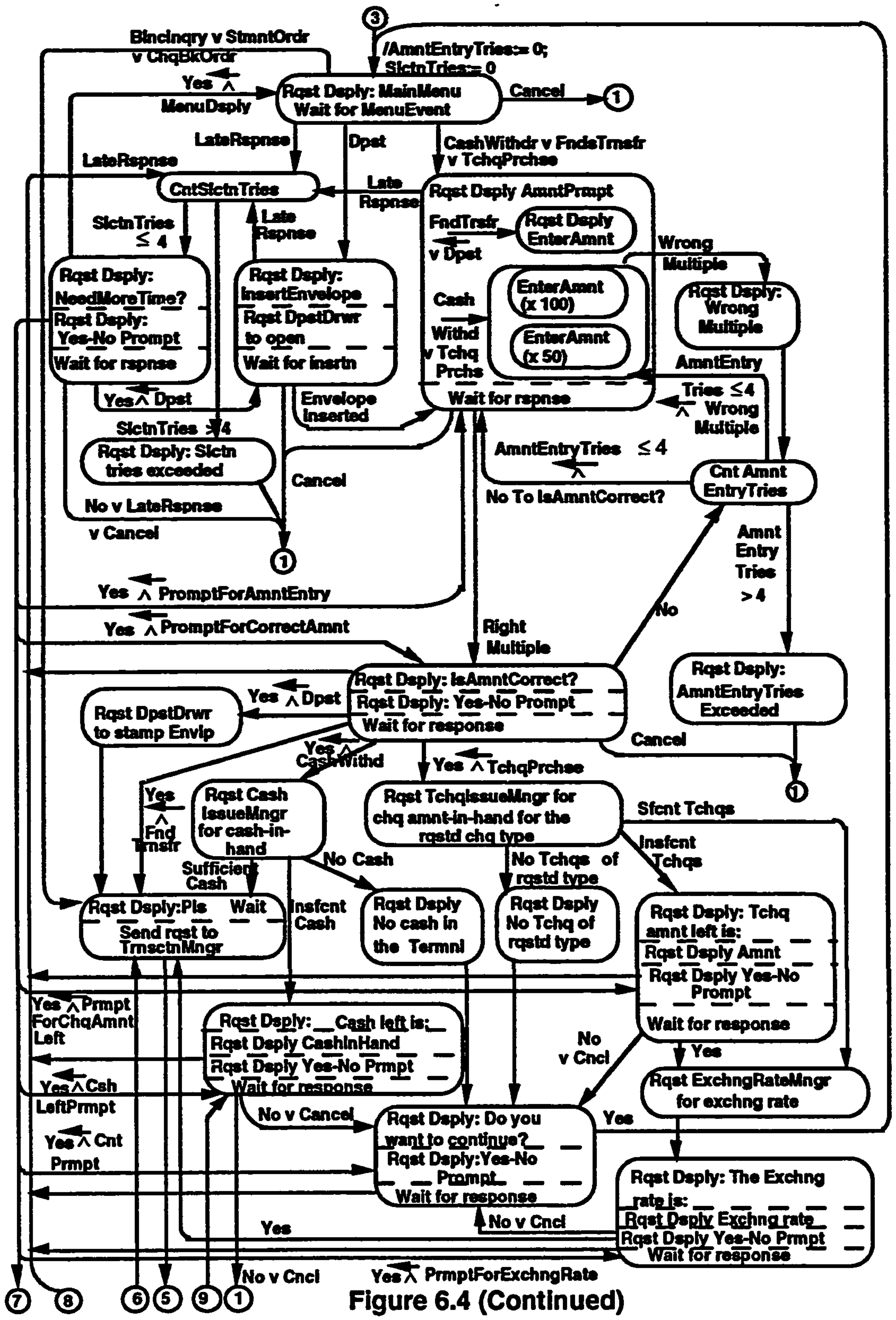


Figure 6.4 (Continued)

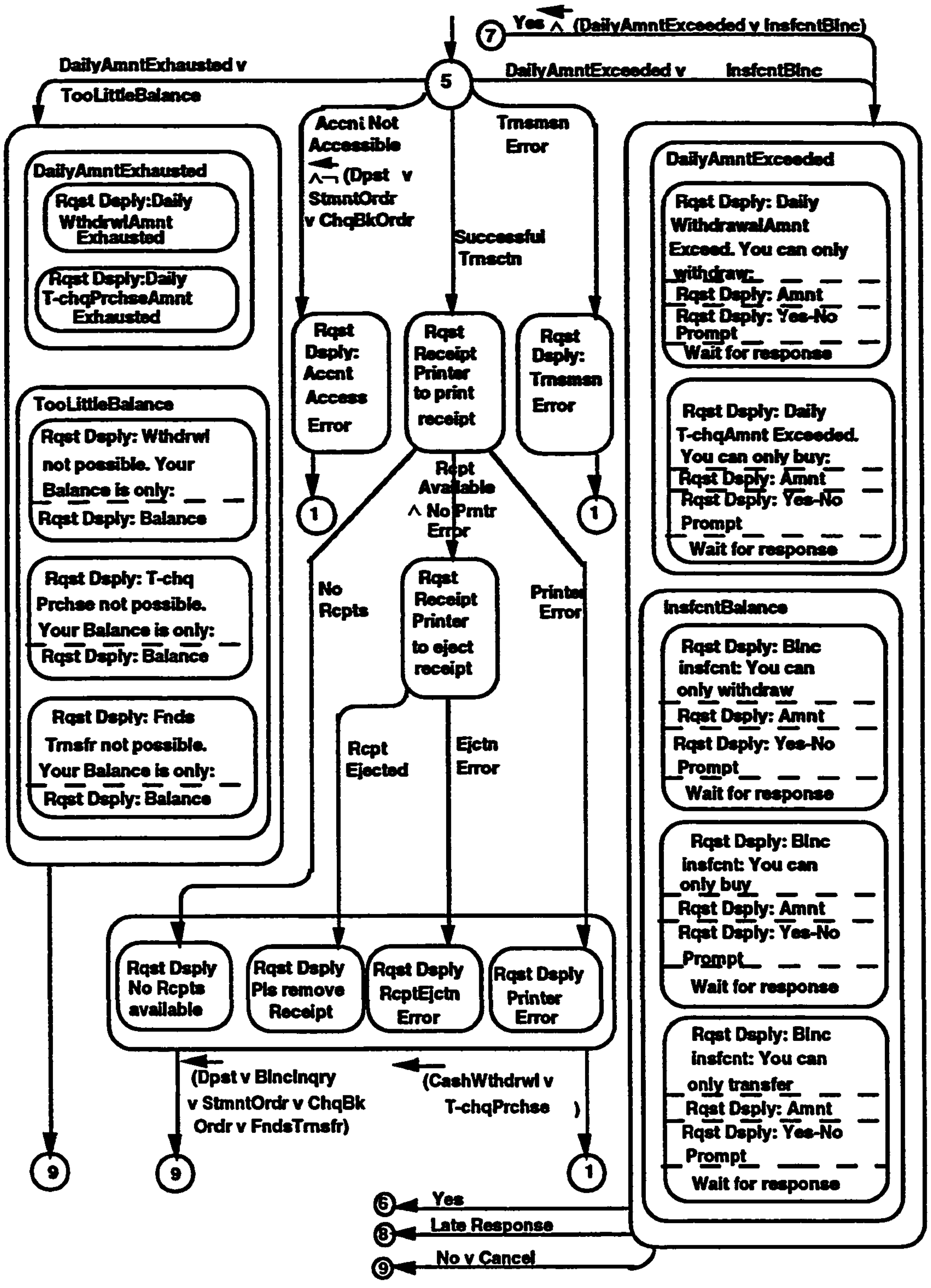

Yes ∧ (DailyAmntExceeded v InsfcntBlnc)
7
5
DailyAmntExhausted v TooLittleBalance
DailyAmntExceeded v InsfcntBlnc
Accnt Not Accessible ∧¬ (Dpst v StmntOrdr v ChqBkOrdr
Trnsmsn Error
Successful Trnsctn
DailyAmntExhausted
Rqst Dsply:Daily WthdrwlAmnt Exhausted
Rqst Dsply:Daily T-chqPrchseAmnt Exhausted
TooLittleBalance
Rqst Dsply: Wthdrwl not possible. Your Balance is only:
Rqst Dsply: Balance
Rqst Dsply: T-chq Prchse not possible. Your Balance is only:
Rqst Dsply: Balance
Rqst Dsply: Fnds Trnsfr not possible. Your Balance is only:
Rqst Dsply: Balance
Rqst Dsply: Accnt Access Error
Rqst Receipt Printer to print receipt
Rqst Dsply: Trnsmsn Error
1
1
Rcpt Available ∧ No Prntr Error
No Rcpts
Printer Error
Rqst Receipt Printer to eject receipt
Rcpt Ejected
Ejctn Error
DailyAmntExceeded
Rqst Dsply: Daily WithdrawalAmnt Exceed. You can only withdraw:
Rqst Dsply: Amnt
Rqst Dsply: Yes-No Prompt
Wait for response
Rqst Dsply: Daily T-chqAmnt Exceeded. You can only buy:
Rqst Dsply: Amnt
Rqst Dsply: Yes-No Prompt
Wait for response
InsfcntBalance
Rqst Dsply: Blnc insfcnt: You can only withdraw
Rqst Dsply: Amnt
Rqst Dsply: Yes-No Prompt
Wait for response
Rqst Dsply: Blnc insfcnt: You can only buy
Rqst Dsply: Amnt
Rqst Dsply: Yes-No Prompt
Wait for response
Rqst Dsply: Blnc insfcnt: You can only transfer
Rqst Dsply: Amnt
Rqst Dsply: Yes-No Prompt
Wait for response
Rqst Dsply No Rcpts available
Rqst Dsply Pls remove Receipt
Rqst Dsply RcptEjctn Error
Rqst Dsply Printer Error
(Dpst v BlncInqry v StmntOrdr v ChqBk Ordr v FndsTrnsfr)
(CashWthdrwl v T-chqPrchse )
9
9
1
6 Yes
8 Late Response
9 No v Cancel

Figure 6.4 (Continued)

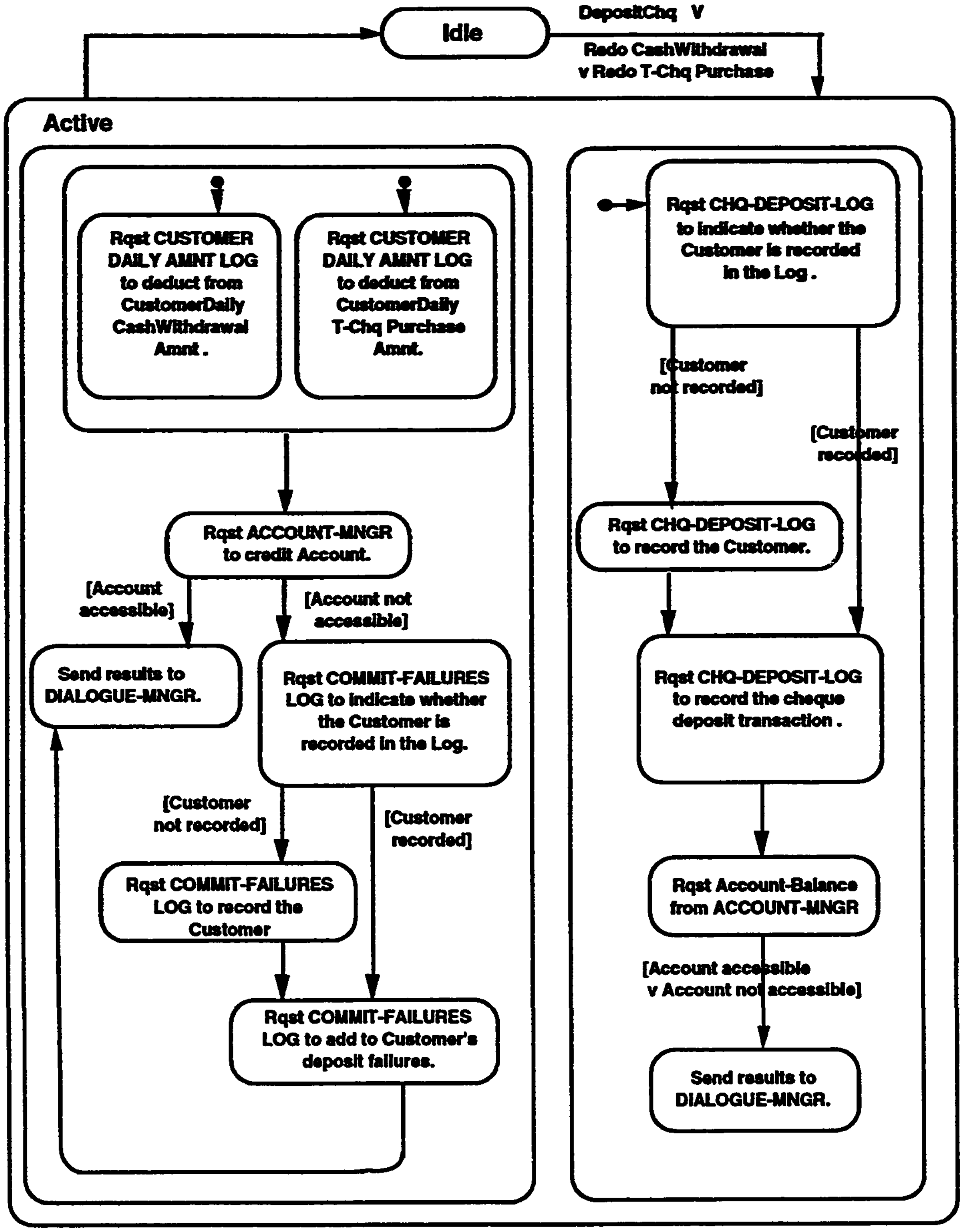
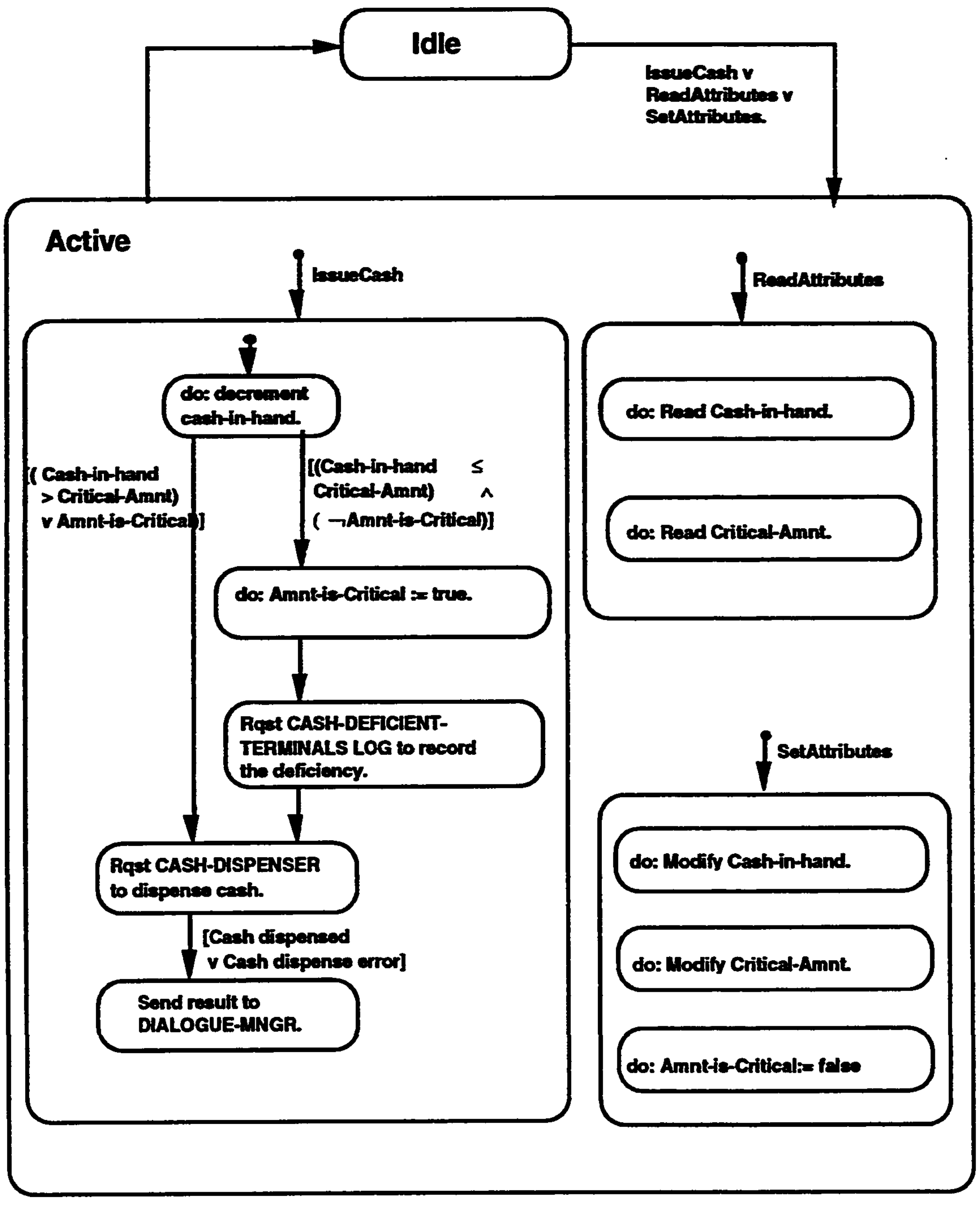
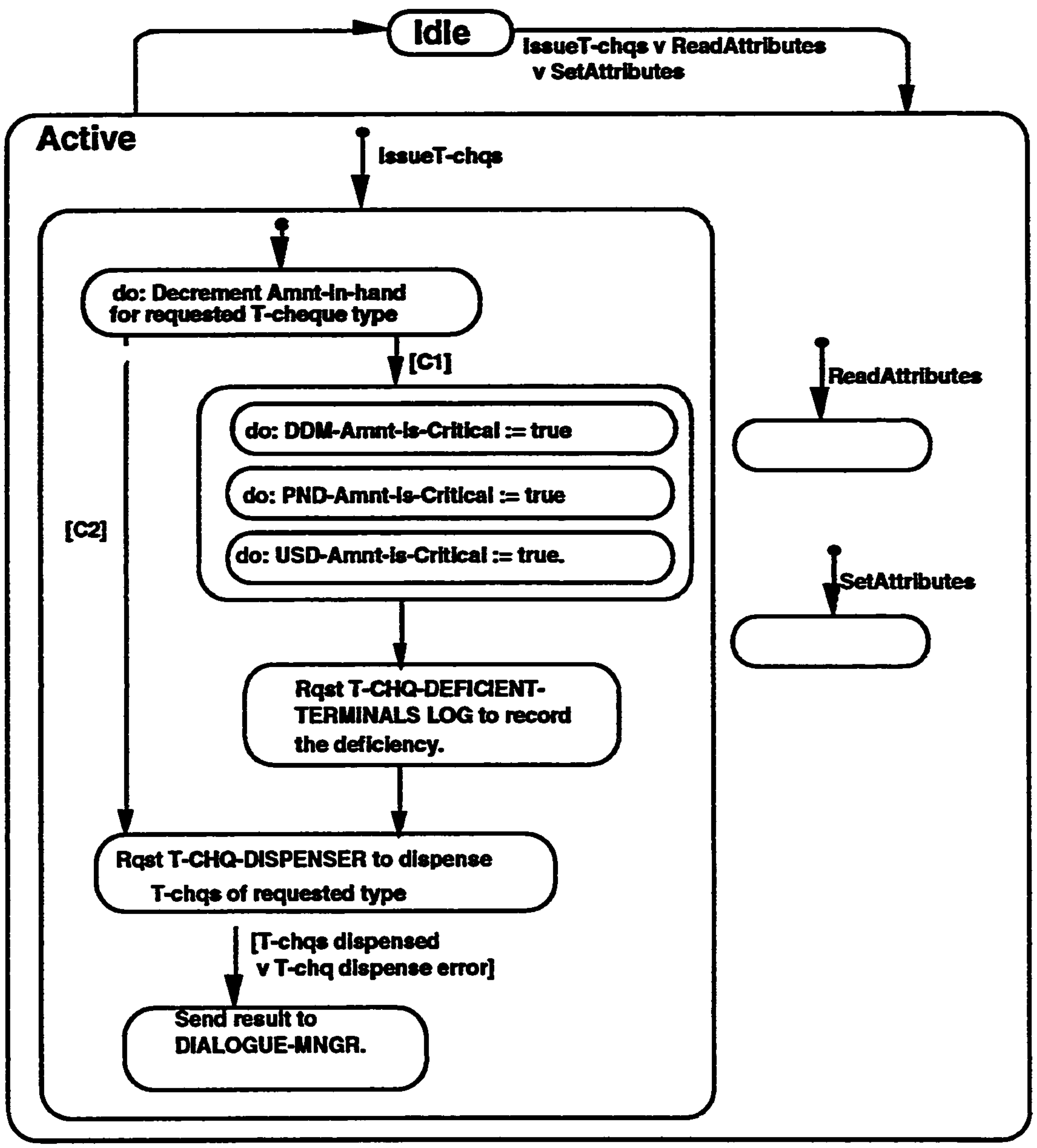

Idle
DepositChq V
Redo CashWithdrawal
v Redo T-Chq Purchase
Active
Rqst CUSTOMER DAILY AMNT LOG to deduct from CustomerDaily CashWithdrawal Amnt.
Rqst CUSTOMER DAILY AMNT LOG to deduct from CustomerDaily T-Chq Purchase Amnt.
Rqst ACCOUNT-MNGR to credit Account.
[Account accessible]
[Account not accessible]
Send results to DIALOGUE-MNGR.
Rqst COMMIT-FAILURES LOG to indicate whether the Customer is recorded in the Log.
[Customer not recorded]
[Customer recorded]
Rqst COMMIT-FAILURES LOG to record the Customer
Rqst COMMIT-FAILURES LOG to add to Customer's deposit failures.
Rqst CHQ-DEPOSIT-LOG to indicate whether the Customer is recorded in the Log.
[Customer not recorded]
[Customer recorded]
Rqst CHQ-DEPOSIT-LOG to record the Customer.
Rqst CHQ-DEPOSIT-LOG to record the cheque deposit transaction.
Rqst Account-Balance from ACCOUNT-MNGR
[Account accessible v Account not accessible]
Send results to DIALOGUE-MNGR.

Figure 6.5: DepositManager statechart



Idle
IssueCash v ReadAttributes v SetAttributes.
Active
IssueCash
ReadAttributes
SetAttributes
do: decrement cash-in-hand.
[( Cash-in-hand > Critical-Amnt) v Amnt-is-Critical)]
[(Cash-in-hand ≤ Critical-Amnt) ∧ ( ¬Amnt-is-Critical)]
do: Amnt-is-Critical := true.
Rqst CASH-DEFICIENT-TERMINALS LOG to record the deficiency.
Rqst CASH-DISPENSER to dispense cash.
[Cash dispensed v Cash dispense error]
Send result to DIALOGUE-MNGR.
do: Read Cash-in-hand.
do: Read Critical-Amnt.
do: Modify Cash-in-hand.
do: Modify Critical-Amnt.
do: Amnt-is-Critical:= false

Figure 6.6: CashIssueManager statechart


Idle
IssueT-chqs v ReadAttributes v SetAttributes
Active
IssueT-chqs
do: Decrement Amnt-in-hand for requested T-cheque type
[C1]
[C2]
do: DDM-Amnt-is-Critical := true
do: PND-Amnt-is-Critical := true
do: USD-Amnt-is-Critical := true.
ReadAttributes
SetAttributes
Rqst T-CHQ-DEFICIENT-TERMINALS LOG to record the deficiency.
Rqst T-CHQ-DISPENSER to dispense T-chqs of requested type
[T-chqs dispensed v T-chq dispense error]
Send result to DIALOGUE-MNGR.

$$C1 = ((\text{USD-Amnt-in-hand} \le \text{USD-Critical-Amnt}) \wedge (\neg \text{USD-Amnt-is-Critical}))$$
$$\vee\ ((\text{PND-Amnt-in-hand} \le \text{PND-Critical-Amnt}) \wedge (\neg \text{PND-Amnt-is-Critical}))$$
$$\vee\ ((\text{DDM-Amnt-in-hand} \le \text{DDM-Critical-Amnt}) \wedge (\neg \text{DDM-Amnt-is-Critical}))$$

$$C2 = (\text{USD-Amnt-in-hand} > \text{USD-Critical-Amnt} \vee \text{USD-Amnt-is-Critical})$$
$$\wedge\ (\text{PND-Amnt-in-hand} > \text{PND-Critical-Amnt} \vee \text{PND-Amnt-is-Critical})$$
$$\wedge\ (\text{DDM-Amnt-in-hand} > \text{DDM-Critical-Amnt} \vee \text{DDM-Amnt-is-Critical})$$

Figure 6.7: TravellersChequeIssueManager statechart

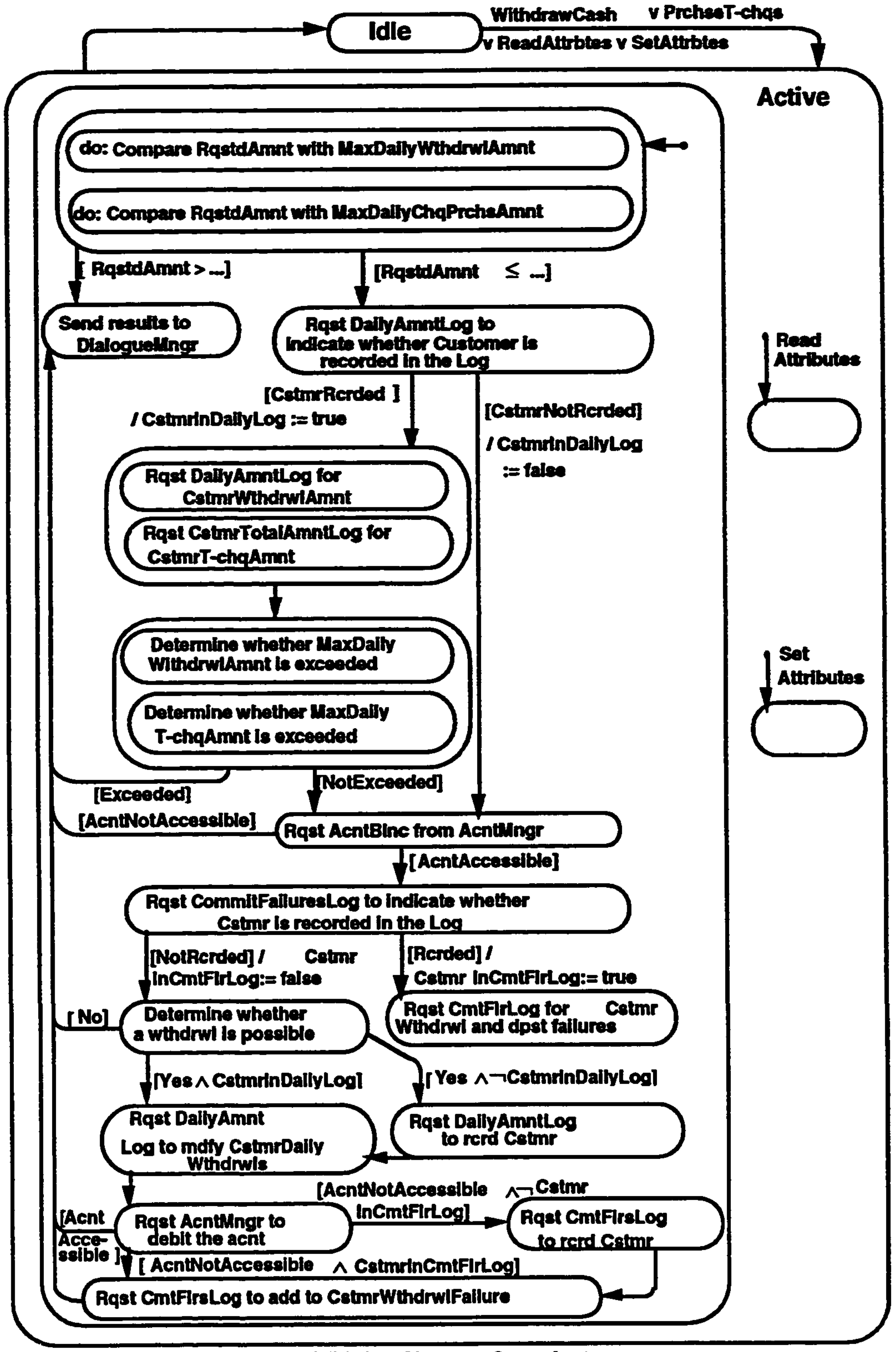

Idle
WithdrawCash v PrchseT-chqs v ReadAttrbtes v SetAttrbtes
Active
do: Compare RqstdAmnt with MaxDailyWthdrwlAmnt
do: Compare RqstdAmnt with MaxDailyChqPrchsAmnt
[ RqstdAmnt > ...]
[RqstdAmnt ≤ ...]
Send results to DialogueMngr
Rqst DailyAmntLog to indicate whether Customer is recorded in the Log
[CstmrRcrded ] / CstmrInDailyLog := true
[CstmrNotRcrded] / CstmrInDailyLog := false
Rqst DailyAmntLog for CstmrWthdrwlAmnt
Rqst CstmrTotalAmntLog for CstmrT-chqAmnt
Determine whether MaxDaily WithdrwlAmnt is exceeded
Determine whether MaxDaily T-chqAmnt is exceeded
[Exceeded]
[NotExceeded]
[AcntNotAccessible]
Rqst AcntBlnc from AcntMngr
[AcntAccessible]
Rqst CommitFailuresLog to indicate whether Cstmr is recorded in the Log
[NotRcrded] / Cstmr InCmtFlrLog:= false
[Rcrded] / Cstmr InCmtFlrLog:= true
Rqst CmtFlrLog for Cstmr Withdrwl and dpst failures
[No]
Determine whether a wthdrwl is possible
[Yes ∧ CstmrInDailyLog]
[ Yes ∧ ¬CstmrInDailyLog]
Rqst DailyAmnt Log to mdfy CstmrDaily Withdrwls
Rqst DailyAmntLog to rcrd Cstmr
[AcntNotAccessible ∧ ¬Cstmr InCmtFlrLog]
[Acnt Accessible ]
Rqst AcntMngr to debit the acnt
Rqst CmtFlrsLog to rcrd Cstmr
[ AcntNotAccessible ∧ CstmrInCmtFlrLog]
Rqst CmtFlrsLog to add to CstmrWthdrwlFailure
Read Attributes
Set Attributes

Figure 6.8: WithdrawManager Statechart

# CHAPTER 7
# CONCLUSIONS AND FUTURE WORK

## 7.1 Conclusions

An object-oriented software design methodology has been developed. It is based on a static- and a dynamic-model. The static-model uses a form of an enhanced entity-relationship model, inheritance-diagrams, and CRC (Class Responsibility Collaboration) cards to capture the static structure of the classes and class relationships. The dynamic-model uses a modified Harel's Statechart notation to capture the internal behaviour of each class and the message passing behaviour among collaborating classes in the system.

The developed design methodology has the following seven steps: exploratory stage, division of the design task into subtasks, extraction of additional subsystems and classes, construction of class interfaces, evaluation and refinement of classes and subsystems, construction of the dynamic-model, and the integration of the design subtasks.

Our methodology differs from others in that it:

- incorporates design-consistency checking. It thus provides a basis for the partial automation of the design process.
- provides a dynamic-model which is easily integrated to the static-model.

## 7.2 Future work

As future work we intend to:

- study more design case studies and rigorously compare our methodology with others with the aim of refining the methodology specifically providing heuristics for identifying subsystems.

- keep abreast with advances in research on metrics for object-oriented design with the aim of introducing more design guidelines in our methodology.
- implement a design-tool which will incorporate design-consistency checking, database support, version control management, and multiple user-interfaces.

## REFERENCES

[Abb83] R.Abbot, " Program Design by Informal English Descriptions," Commun. ACM, Vol. 26, No. 11, p. 884, Nov. 1983.

[AbH87] S.Abiteboul, R.Hull, " IFO: A Formal Semantics Database Model," ACM Trans. Database Systems, Vol.12, No.4, Dec. 1987, pp. 525-565.

[Agh86] G.Agha, " An Overview of Actor Languages," SIGPLAN NOTICES, Vol.21, No.10, 1986, pp. 58-67.

[AkT88] M.Aksit, A.Tripathi, " Data Abstraction Mechanisms in Sina/st," OOPSLA'88 Conf. Proc., Sandiego, California, Sept. 25-30 1988; ACM SIGPLAN, Vol.23, No.11, Nov. 1988, pp. 267-276.

[Ala88] B.Alabiso, " Transformation of Data Flow Analysis Model to Object-Oriented Design," OOPSLA'88 Conf. Proc., Sandiego, California, Sept. 25-30 1988; ACM SIGPLAN, Vol.23, No.11, Nov. 1988, pp. 335-353.

[ArY90] J.L.Archibald, K.C.B.Yakemovic eds. , " Structured Analysis and Object- Oriented Analysis," Panel in OOPSLA / ECOOP'90 Addendum to the Proc., Ottawa, Canada, 21-25 Oct. 1990, pp. 15-17.

[Bai89] S.Bailin, "An Object-Oriented Requirements Specification Method," Commun. ACM, Vol. 32, No. 5, pp. 608-623, May 1989.

[Bar91] G.Barbedette, " Schema Modifications in LISPO2 Persistent Object-Oriented Language," ECOOP'91 Conf. Proc., 15-19 July 1991, Geneva, Switzerland, P.America (Ed.), pp. 77-96, Springer-Verlag, Berlin Heidelberg, 1991.

[BCG+87] J.Banarjee at. al, "Data Model Issues for Object-Oriented Applications," ACM Trans. Office Info. Syst., Vol. 5, No. 1, Jan. 1987, pp. 3-26.

[BeC89] K.Beck, W.Cunningham, " A Laboratory for Teaching Object-Oriented Thinking," OOPSLA'89 Conf. Proc., 1-6 Oct. 1989, New Orleans, Louisiana; SIGPLAN NOTICES Vol.24, No.10, pp. 1-6, Oct. 1989.

[Bob83] D.G.Bobrow, The LOOPS Manual: A Data and Object-Oriented Programming System for Interlisp, Knowledge-Based VLSI Design Group Memo KB-VLSI-81-13, Xerox PARC, Palo Alto, California, 1983.

[Boe86] B.Boehm, " A Spiral Model of Software Development and Enhancement," ACM SIGSOFT Softw. Engg. Notes, Vol.11, No.4, pp. 14-24, Aug. 1986.

[Boo83] G.Booch, Software Engineering with Ada, Benjamin/Cummings, California, 1983.

[Boo86] G.Booch, " Object-Oriented Development," IEEE Trans. Softw. Eng. , Vol. SE-12, No. 2, pp. 211-222, Feb. 1986.

[Boo91] G.Booch, Object-Oriented Design, Benjamin/Cummings, Readwood City, California, 1991.

[Bro91] D.Brookman, " SA/SD vs OOD, " ACM Ada Letters, Vol. XI, No. 9, pp. 96-99, Nov. /Dec. 1991.

[BuW90] A.Burns, A.Wellings, Real-Time Systems and their Programming Languages, Adison-Wesley, Reading, Massachusetts, 1990.

[Che76] P.Chen, " The Entity-Relationship Model : Toward a Unified View of Data," ACM Trans. Database Systems, Vol.1, No.1, March 1976, pp. 9-36.

[ChK91] S.Chidamber, C.Kemerer, " Towards a Metrics Suite for Object-Oriented Design," OOPSLA'91 conf. Proc., 6-11 Oct. 1991, Phoenix, Arizona; SIGPLAN NOTICES, Vol.26, No.11, Nov. 1991, pp. 197-211.

[Coi87] P.Cointe, " Metaclasses are First Class: The ObjVlisp Model," OOPSLA'87 Conf. Proc., Orlando, Florida, Oct. 1987; SIGPLAN NOTICES, Vol.22, No.12, pp. 156-167.

[CoY90] P.Coad, E.Yourdon, Object-Oriented Analysis, Yourdon Press/Prentice- Hall Englewood Cliffs, NJ , 1990.

[CoY91] P.Coad, E.Yourdon, Object-Oriented Design, Yourdon Press/Prentice-Hall Englewood Cliffs, NJ , 1991.

[Cox86] B.J.Cox, Object-Oriented Programming : An Evolutionary Approach, Addison-Wesley, Reading, MA, 1986.

[Den91] Dennis de Champeaux, " Object-Oriented Analysis and Top-Down Software Development," ECOOP'91 Conf. Proc., 15-19 July 1991, Geneva, Switzerland, P.America (Ed.), pp. 360-375, Springer-Verlag, Berlin Heidelberg, 1991.

[DeZ91] C.Delcourt, R.Ziccari, "The Design of an Integrity Consistency Checker (ICC) for an Object-Oriented Database System," ECOOP'91 Conf. Proc., 15-19 July 1991, Geneva, Switzerland, P.America (Ed.), pp. 97-117, Springer-Verlag, Berlin Heidelberg, 1991.

[Duc90] R.Ducoumau, Y3: An Overview, SEMA Group, Montronge, 1990.

[Dug91] P.Dugerdil, " Inheritance Mechanisms in the OBJLOG Language: Multiple Selective and Multiple Vertical with Points of View, In Inheritance Hierarchies in Knowledge Representation and Programming Languages, pp. 245-256, M.Lenzerini, D.Nardi, M.Simi (eds.), John Wiley & Sons Ltd., Chichester, West Sussex, 1991.

[ElN89] R.Elmasri, S.Navathe, Fundamentals of Database Systems, Benjamin/Cummings, Readwood City, California, 1989.

[ElS90] M.Ellis, B.Stroustrup, The Annotated C++ Reference Manual, Addison-Wesley, Reading, Massachusetts, 1990.

[Fir91] D.Firesmith, " Structured Analysis and Object-Oriented Development are not Compatible." ACM Ada Letters, Vol.XI, No.9, Nov./Dec. 1991, pp. 56-66.

[Gib90] E.Gibson, " Objects - Bom and Bred," BYTE, Oct. 1990, pp. 245-254.

[GLP90] K.Gorlen, S.Orlow, P.Plexico, Data Abstraction and Object-Oriented Programming in C++, John Wiley and Sons, New York, 1990.

[GoA90] S.Gossain, B.Anderson, " An Iterative-Design Model for Reusable Object-Oriented Software," OOPSLA/ECOOP'90 Conf. Proc., 21-25 Oct. 1990, Ottawa, Canada; SIGPLAN NOTICES Vol.25, No.10, Oct. 1990, pp. 12-27.

[GoR83] A.Goldberg, D.Robson, Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, Massachusetts, 1983.

[GoR89] A.Goldberg, D.Robson, Smalltalk-80: The Language, Addison-Wesley, Reading, Massachusetts, 1989.

[Har88] D.Harel, " On Visual Formalisms," Commun. ACM, Vol.31, No.5, May 1988, pp. 514-530.

[HaC91] F.Hayes,D.Coleman, " Coherent Models for Object-Oriented Analysis," OOPSLA'91 Conf. Proc., 6-11 Oct. 1991, Phoenix, Arizona; SIGPLAN NOTICES, Vol.26, No.11, Nov. 1991, pp. 171-183.

[HOH91] M.Hull, P.O'Donoghue, B.Hagan, " Development Methods for Real-Time Systems," The Computer Journal, Vol.34, No.2, 1991, pp. 164-172.

[Hyo91] Hyoung-Joo Kim, "Algorithmic and Computational Aspects of Object-Oriented Schema design," in Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, R.Gupta, E.Horowitz (eds.), Prentice Hall, 1991.

[Jal89] P.Jalote, " Functional Refinement and Nested Objects for Object-Oriented Design," IEEE Trans. Softw. Engg., Vol.15, No.3, March 1989, pp.246-270.

[KBG89] W.Kim, E.Bertino, J.Garza, " Composite Objects Revisited," Proc. ACM SIGMOD 1989, pp. 337-347.

[Kee89] S.E.Keen, Object-Oriented Programming in Common Lisp: A Programmers Guide to CLOS, Addison Wesley, Reading, Massachusetts, 1989.

[Kim90] W.Kim, Introduction to Object-Oriented Databases, The MIT Press, Massachusetts, 1990.

[Kno89] N.Knolle, " Why Object-Oriented User Interface Tools Are Better," Journal of Object-Oriented Programming, Vol.2, No.4, 1989, pp. 63-67.

[Law88] D.Law, Methods for Comparing Methods: Techniques in Software Development, NCC Publications, Manchester, England, 1988.

[LeH90] B.S.Lerner, A.Habermann, "Beyond Schema Evolution to Database Reorganization," OOPSLA/ECOOP'90 Conf. Proc., 21-25 Oct. 1990, Ottawa, Canada, as SIGPLAN NOTICES Vol.25, No.10, Oct. 1990, pp. 67-76.

[LHR88] K.Lierberherr, I.Holland, A.Riel, " Object-Oriented Programming: An Objective Sense of Style, " OOPSLA'88 Conf. Proc., Sandiego, California, Sept. 25-30 1988; ACM SIGPLAN, Vol.23, No.11, Nov. 1988, pp. 323-334.

[Lie86] H.Lieberman, " Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems," OOPSLA'86 Conf. Proc., Sept. 29- Oct. 2, 1986, Portland, Oregon; SIGPLAN NOTICES, Vol.21, No.11, Nov. 1986, pp. 214-223.

[Lie87] H.Lieberman, Concurrent Object-Oriented Programming in Act1, in Object-Oriented Concurrent Programming, pp. 9-36, A.Yonezawa, M.Tokoro (eds), The MIT Press, 1987.

[LiH89] K.Lierberherr, I.Holland, " Assuring Good Style for Object-Oriented Programms," IEEE Softw., Sept. 1989, pp. 38-48.

[LVC89] M.Linton, J.Vlissides, P.Calder, " Composing User Interfaces with InterViews," Computer, Vol.22, No.2, 1989, pp. 8-22.

[LiR89] K.Lieberherr, A.Riel, " Contributions to Teaching Object-Oriented Design and Programming," OOPSLA'89 Conf. Proc., 1-6 Oct. 1989, New Orleans, Louisiana; SIGPLAN NOTICES Vol.24, No.10, pp. 371-380, Oct. 1989.

[Mey88] B.Meyer, " Eiffel: A Language and Environment for Software Engineering," The Journal of Systems and Software, Vol.8, 1988, pp. 199-246.

[Mey90] B.Meyer, " Tools for the new culture: Lessons from the design of the Eiffel libraries," Commun. ACM, Vol.33, No.9, Sept. 1990, pp. 68-89.

[MiR89] N.Minsky, D.Rozenshtein, " Controllable Delegation: An Exercise in Law-Governed Systems," OOPSLA'89 Conf. Proc., 1-6 Oct. 1989, New Orleans, Louisiana; SIGPLAN NOTICES Vol.24, No.10, pp. 371-380, Oct. 1989.

[MNC+91] G.Masini, A.Napoli, D.Golnet, D.Leornad,K.Tombre, Object-Oriented Languages, Academic Press, London, 1991.

[Moo86] D.Moon, " Object-Oriented Programming with Flavors," OOPSLA'86 Conf. Proc., Sept. 29- Oct. 2, 1986, Portland, Oregon; SIGPLAN NOTICES, Vol.21, No.11, Nov. 1986, pp. 1-8.

[Moo89] D.Moon, " The Common LISP Object-Oriented Programming Standard System," in Object-Oriented Concepts, Applications, and Databases, W.Kim, F.Lochovsky (eds.), Addison Wesley, 1989.

[Myn90] B.Mynatt, Software Engineering with Student Project Guidance, Prentice-Hall, Englewood, Cliffs, New Jersey, 1990.

[Ner92] Jean-Mark Nerson, : " Applying Object-Oriented Analysis and Design," Commun. ACM, Vol.35, No.9, Sept. 1992, pp. 63-74.

[Nie87] O.M.Nierstrasz, "Active Objects in Hybrid," OOPSLA'87 Conf. Proc., Orlando, Florida, Oct. 1987; SIGPLAN NOTICES, Vol.22, No.12, pp. 243-253.

[RBP+91] J.Rumbaugh, M.Blaha, W.Premerlani, F.Eddy, W.Lorensen, Object-Oriented Modelling and Design, Prentice-Hall, Englewood, Cliffs, New Jersey, 1991.

[RuG92] K.Rubin, A.Goldberg, " Object Behaviour Analysis," Commun. ACM, No.9, Sept. 1992, pp. 48-62.

[SaP90] A.Sage, J.Palmer, Software Systems Engineering, John Wiley & Sons, Inc., New York, 1990.

[SCB+86] C.Schaffert, T.Cooper, B.Bullis, M.Kilian, C.Wilpolt, " An Introduction to Trellis/Owl," OOPSLA'86 Conf. Proc., Sept. 29- Oct. 2, 1986, Portland, Oregon; SIGPLAN NOTICES, Vol.21, No.11, Nov. 1986, pp. 9-16.

[Sei89] E.Seidewitz, " General Object-Oriented Software Development: Background and Experience." Journal of Systems Software, Vol.9, 1989, pp. 95-108.

[SeS86] E.Seidewitz, M.Stark, " Towards a General Object-Oriented Software Development Methodology," in Proc. 1st Conf. on Ada Programming Language Applications for the NASA Space Station, June 1986, pp. D.4.6.1-D.4.6.14.

[ShM88] S.Shlaer, S.Mellor, Object-Oriented Systems Analysis, Yourdon Press, Englewood, Cliffs, New Jersey, 1988.

[Shu91] K.Shumate, " Structured Analysis and Object-Oriented Design are Compatible, " Ada Letters, Vol.XI, No.4, pp. 78-90, May/June 1991.

[Sny86] A.Snyder, "Common Objects: An Overview," SIGPLAN NOTICES, Oct. 1986, pp. 19-28.

[Ste87] L.Stein, "Delegation is Inheritance," OOPSLA'87 Conf. Proc., Oct. 1987; SIGPLAN NOTICES, Vol.22, No.12, pp. 138-146.

[TSK90] C.Tomlinson, M.Scheevel, W.Kim, " Sharing and Organisation Protocols in Object-Oriented Systems," Journal of Object-Oriented Programming, Vol.2, No.4, 1990, pp.25-36.

[UnS87] D.Ungar, R.Smith, " Self: The Power and Simplicity," OOPSLA'87 Conf. Proc., Oct. 1987; SIGPLAN NOTICES, Vol.22, No.12, pp. 227-242.

[Wal91] N.Walters, " An Ada Object-Based Analysis and Design Approach," Ada Letters, Vol.XI, No.5, pp. 62-78, July/Aug. 1991.

[WaM85] P.Ward, S.Mellor, Structured Development for Real-Time Systems, Vol.1 , Yourdon Press/Prentice-Hall, Englewood, Cliffs, New Jersey, 1985.

[WiJ90] R.Wirfs-Brock, R.Johnson, " Surveying Current Research in Object-Oriented Design," Commun. ACM, vol. 33, No. 9, pp. 104-124, Sept. 1990.

[WiW89] R.Wirfs-Brock, B.Wilkerson, " Object-Oriented Design: A Responsibility- Driven Approach, " OOPSLA'89 Conf. Proc., 1-6 Oct. 1989, New Orleans, Louisiana; SIGPLAN NOTICES Vol.24, No.10, pp. 71-75, Oct. 1989.

[WPM89] A.Wasserman, P.Pircher, R.Muller, " An Object-Oriented Structured Design Method for Code Generation," ACM SIGSOFT Softw. Engg. Notes, Vol.14, No.1, Jan. 1989, pp. 32-55.

[WPM90] A.Wasserman, P.Pircher, R.Muller, " The Object-Oriented Structured Design Notation for Software Design Representation," Computer, March 1990, pp. 50-63.

[WRS90] D.Wilson, L.Rosenstein, D.Shafer, Programming with McApp, Addison-Wesley, Reading, Massachusetts,1990.

[WWW90] R.Wirfs-Brock, B.Wilkerson, L.Wiener, Designing Object-Oriented Software, Prentice-Hall, Englewood, Cliffs, New Jersey, 1990.

[YBS86] A.Yonezawa, J.Briot, E.Shibayama, " Object-Oriented Concurrent Programming in ABCL/1," OOPSLA'86 Conf. Proc., Sept. 29- Oct. 2, 1986, Portland, Oregon; SIGPLAN NOTICES, Vol.21, No.11, Nov. 1986, pp. 258-268.

[Yon90] A.Yonezawa (ed.), ABCL: An Object-Oriented Concurrent System, The MIT Press, Cambridge, Massachusetts, 1990.