Structuring Fault-Tolerant Object-Oriented Systems Using Inheritance and Delegation

Cecília Mary Fischer Rubira

Ph.D. Thesis

Department of Computing Science

University of Newcastle upon Tyne

October 1994

Abstract

Many entities in the real world that a software system has to interact with, e.g., for controlling or monitoring purposes, exhibit different *behaviour phases* in their lifetime, in particular depending on whether or not they are functioning correctly. That is, these entities exhibit not only a normal behaviour phase but also one or more abnormal behaviour phases associated with the various faults which occur in the environment. These faults are referred to as *environmental faults.* In the object-oriented software, real-world entities are modeled as objects. In a class-based object-oriented language, such as C++, all objects of a given class must follow the same *external behaviour*, i.e., they have the same interface and associated implementation. However this requires that each object permanently belong to a particular class, imposing constraints on the mutability of the behaviour for an individual object. This thesis proposes solutions to the problem of finding means whereby objects representing real-world entities which exhibit various behaviour phases can make corresponding changes in their own behaviour in a clear and explicit way, rather than through status-checking code which is normally embedded in the implementation of various methods.

Our proposed solution is (i) to define a hierarchy of different subclasses related to an object which corresponds to an external entity, each subclass implementing a different behaviour phase that the external entity can exhibit, and (ii) to arrange that each object forward the execution of its operations to the currently appropriate instance of this hierarchy of subclasses. We thus propose an object-oriented approach for the provision of environmental fault tolerance, which encapsulates the abnormal behaviour of "faulty" entities as objects (instances of the above mentioned subclasses). These abnormal behaviour variants are defined statically, and runtime access to them is implemented through a *delegation* mechanism which depends on the current phase of behaviour. Thus specific reconfiguration changes at the level of objects can be easily incorporated to a software system for tolerating environmental faults.

Key Words - Object-Oriented Programming, Delegation, Dependability, Fault Tolerance, Exception Handling.

BLANK PAGE
IN
ORIGINAL

"To my
son
Pedro Gabriel."

BLANK PAGE
IN
ORIGINAL

Acknowledgements

I am deeply indebted to my supervisor Brian Randell for his consistent support, advice and criticism, and many illuminating discussions and meetings over the years. I also would like to thank my collegues Robert Stroud, Jie Xu, Laurent Blair, Alexander Sascha Romanoviski, and Zhixue Wu for the inumerous discussions about object-oriented programming and fault tolerance.

There are others also whose help has been absolutely invaluable. In special, I would like to thank Shirley Craig for helping me so much with the references of my work and for being so kind and patient. I am also grateful to Anke Jackson and Christine Davies for being so supportive in many times that I have asked for help. Trevor Kirby is also thanked for his efforts in the experiments with the train set.

I am grateful to the Conselho Nacional de Desenvolvimento Científico e Tecnológico - CNPq (Brazil - grant no. 200198/90.4), and also to the ESPRIT Basic Research Action entitled "Predictable Dependable Computing Systems" - PDCS (research project 6362) for financial support.

I am grateful, too, to all those wonderful friends who are part of my life. Maeve de Mello, Gilson P. Manfio, Luiz E. Buzato, Rogério de Lemos, Roberta Araruna, Teresa Gomes, and Rosemary Nicholson deserve special mention for constant encouragement and support. Last, but most of all, my thanks to my family for being the people they are; in special, to my mother and father.

BLANK PAGE
IN
ORIGINAL

Contents

List of Figures

List of Tables

Chapter 1

Introduction

> "The fact, then, that many complex systems have a nearly decomposable, hierarchic structure is a major facilitating factor enabling us to understand, to describe, and even to see such systems and their parts.
>
> Or perhaps the proposition should be put the other way round. If there are important systems in the world which are complex, they may to a considerable extent escape our observation and our understanding. Analysis of their behaviour would involve such detailed knowledge and calculation of the interactions of the elementary parts that it would be beyond our capacities of memory or computation."[139]

Most real-world computer systems are extremely complicated, as is evidenced by the intricacy of their hardware and software implementations, and they have become even more complicated since the requirements of new applications have become more demanding: high size and complexity, high reliability and high performance. It is sometimes possible, even desirable, to ignore this complexity, but doing so does not make it go away because it will certainly appear somewhere else. Software applications that deal with complicated problems must cope with this complexity somewhere. The complexity of most computing systems is principally found in their software, since hardware systems essentially have to be comparatively simple, with limited and constrained interconnections, and simple interactions between components[91]. Hence many ongoing investigations are concerned with mastering software complexity.

More than 25 years ago, the term *software crisis* was coined at the 1968 NATO Software Engineering Conference[116] to indicate that software was already scarce, expensive, of insufficient quality, hard to schedule, and nearly impossible to man-

age. The lack of new technologies that address new methods and tools which guarantee good quality in software development is undeniable. The crucial aim of software development is to build quickly, cheaply, flexibly and reliably.

Since the early 80s, the object-oriented paradigm has been considered as one of the most significant developments in software programming, comparable only to the ideas of structured programming in the early 70s. The object-oriented paradigm is well suited for structuring a wide variety of complex systems, notably providing a sound basis to develop applications that are easy to maintain and reuse. Object-oriented design techniques focus on entities and abstractions that relate to the problem domain and object-oriented languages can directly capture the structure of these abstractions in a solution domain.

In many real-world situations entities of the problem domain may change their behaviour[1] due to a variety of phenomena, e.g., equipment failure or malfunction, personnel errors, sensor failures. Assuming that the problem domain could be fully understood at the outset, one could identify many different ways in which its entities could behave, and also categorise the different kinds of changes that occur in the problem domain. Such changes can be classified in many different ways, such as frequent vs. infrequent, desired vs. undesired, or predictable vs. unpredictable. In this research, we are particularly interested in behaviour changes caused by (relatively) infrequent, undesired and predictable changes of the environment. We refer to such changes of the environment as *environmental faults*. Thus, in a problem domain, various sources of complexity can be found, but the primary focus of this work concentrates on environmental faults, and the frequent need to cope with them as an unvoidable source of complexity, but one which nevertheless has to be minimised. This thesis demonstrates how object-oriented techniques can be used to implement software components that support environmental fault tolerance.

In the object-oriented software, entities in the real world are modeled as objects[156]. An object has a time invariant identity that is unique with respect to other objects. Objects sharing the same structural and operational characteristics are classified into classes. A class is a way of describing the structure and the behaviour[2] of a family of objects. Inheritance is a mechanism for deriving new

[1] Here we mean behaviour in the sense defined by Booch[20], namely how an entity acts and reacts, in terms of its state changes and message passing.

[2] In object-oriented languages, it can be assumed that all properties of an object will be encapsulated within it, and, furthermore, this encapsulation will be protected behind an *abstract interface*. In the context of abstract data type theory, the term *behaviour* is used with the restricted meaning of denoting this abstract interface, which normally consists of a set of applicable operations. An operation is some action that one object performs upon another to elicit a reaction. Type is then denotable in terms of this behaviour, irrespective of the under-

classes from existing classes by a process of refinement. A derived class inherits the data representation and operations of its parent class but may selectively add new operations, extend the data representation or override the implementation of existing operations. Thus programming is achieved by class definitions and hierarchy creation, in which properties are transmitted from superclasses to subclasses by the concept of inheritance. Object-oriented software construction integrates established software engineering principles like data abstraction[96], encapsulation[141] and strong typing[99].

For purposes of this research, we restrict the inheritance relationship to the so-called *behavioural* inheritance model[96] that is increasingly being accepted as the model that supports a subtyping/conformance relationship. In this model, a derived class is designed by including the specification of the parent class as a subset of the specification of the derived class. However, the requirements of this model go beyond the textual matching of operation names. It is important that overriding of operations by a derived class be performed in a disciplined manner so that the derived class still realises the same basic abstraction as the parent class. Meyer has proposed a contract model for inheritance, known as *design by contract*[108], in which the behaviour of a class is constrained by pre- and post-conditions on its operations and an invariant over its internal state. Derived classes are obliged to conform to the specification of their parent classes in the sense that the invariant for the derived class must preserve the invariant for the parent class, while each operation of the derived class that replaces an operation of the parent class must have a weaker pre-condition and a stronger post-condition. These two requirements produce an inheritance hierarchy in which the subclasses are subtype compatible with their parents and polymorphic substitution of instances of derived classes for instances of parent classes is possible without affecting the correctness of the overall system. In what follows, we refer to this model of inheritance as *restricted inheritance* which encompasses both notions of behavioural inheritance and "design by contract" methodology.

Even when all possible behaviour changes of the entities related to the problem domain have been envisaged at the outset, it can be desirable to represent such changes explicitly in the solution domain, rather than simply encode them into the object's methods. Thus, if we have chosen to identify the behaviour of an entity of a problem domain as involving several distinct *behaviour phases*, transitions between which are caused by occurrence and disappearance of environmental faults, so we would wish to represent such behaviour phases and phase transitions explicitly in the solution domain. The behaviour phase that an entity is in is

---

lying implementation or data structures which might be subject to change[26]. From now on, the term behaviour denotes this meaning.

known as its *current behaviour phase.*

According to the scope of this thesis, we classify the behaviour of an object in the solution domain, corresponding to such problem domain entity, into: a *normal* behaviour phase, when it is representing an external entity which is behaving normally, and one or more kinds of *abnormal* behaviour phases, which we intend to correspond to the various kinds of faulty behaviour by the external entity.

An object in the solution domain is referred to as *faulty* if its behaviour is altered due to the occurrence of environmental faults in the entities of the problem domain. When the current behaviour phase of an object is changed, for instance, from a normal to an abnormal behaviour phase, the object provides the adequate fault treatment by changing the implementation of its abstract interface.

In this thesis, we suggest that transitions between these various behaviour phases of an object be accomplished by changing the implementation of the object's abstract interface (i.e., of one or more of its operations), and that the behaviour phases associated with the different behaviour phases be explicitly represented as objects.

1.1 System Structure and a Fault Taxonomy

Complexity is one of the major causes of unreliable software. In a problem domain, various sources of complexity can be found, but the primary focus of this thesis is to concentrate on faults (or major changes) in the environment, and the frequent need to cope with them as an unvoidable cause, but one which nevertheless has to be minimised. In this thesis, we follow the terminology defined by Lee and Anderson[91]. More specifically, we regard a system as any identifiable mechanism which maintains a pattern of behaviour at an interface between the mechanism and its environment. An *interface* is simply a place of interaction between two systems. A system is said to interact with its *environment*, and responds to stimuli at the interface between the system and the environment. The environment of a system is another system which provides input to and receives output from the first system; thus the system can provide service in response to request from the environment. In the scope of this thesis, we are mainly considering a software system as our solution domain which is surrounded by an environment that contains the problem domain. By way of example, if we consider a process control application, according to our terminology, the system can refer to a computer-based controller while the environment would be the physical process or plant which is maintained or monitored by the controller.

The above definition is concerned with the external characteristics of the system. However, it is also important to have an abstract model of its internal structure. In this sense, we define a *system* as consisting of a set of interacting components together with a design which prescribes and controls the pattern of interaction. An erroneous value in the state of a component or in the design of a system is referred to as a *fault* in the system. More specifically, a *component fault* is an error in the internal state of a component whereas a *design fault* is an error in the state of the design. More generally, faults may also occur in the environment of a system, and we use the term *environmental fault* to refer to an error in the state of the environment. Design faults are unpredictable, their manifestation is unexpected and they produce unanticipated errors. In contrast, component faults (particularly those in hardware components of the system) can often be predicted, their manifestation is expected and they generate errors which can be anticipated. In a similar way, environmental faults, specially those in physical entities of the environment, are frequently predictable.

With such a view of a system structure and considering the distinct place of fault occurrence, we classify faults into three categories: *environmental*, *design* and *hardware*. As stated previously, this works concentrates mainly on dealing with environmental faults as an unvoidable source of complexity in the process of designing and implementing dependable software systems.

1.2 Thesis Overview

The major goal of this work is to apply object-oriented techniques for structuring complex object-oriented applications, and relate them to the problems of improving quality and dependability of large computer applications. The object-oriented approach was chosen because of the benefits derived from the following concepts, such as, data abstraction, data encapsulation, inheritance, and polymorphism.

More specifically, the problem that we propose to solve concerns the real-world entities that a software system has to interact with, which have not only a normal behaviour phase but also one or more abnormal behaviour phases associated with the occurrence of different environmental faults. However, when one starts allowing for all the abnormal behaviour phases of such entities, the problem domain can become extremely complex. As a consequence, the object-oriented model becomes correspondingly larger and more complicated as more and more abnormal cases are considered. Thus we need to find ways of structuring such software systems so that this complexity can be kept under control.

We could simply bury the different behaviour phases associated to an external entity in the implementation of the operations of the corresponding object in the solution domain. However, such an implementation can become quite complicate and obscure depending on the number of abnormal behaviour phases that the external entity exhibits. It should be ideal if such behaviour phases could be represented by different implementations of the object's interface, each one corresponding to a different behaviour phase.

In class-based systems, objects of a given class must follow the same behaviour, i.e., they have a same interface and associated implementation. Moreover it is required that each object permanently belongs to a particular class, which imposes constraints on the mutability of the behaviour for an individual object. In this thesis, as mentioned above, we define the behaviour of an object as involving several distinct behaviour phases, each behaviour phase representing a faulty behaviour of the external entity, the transitions between such phases being caused by the occurrence and disappearance of environmental faults. So all objects of a given class follow the same external behaviour; however, at any particular moment, different instances of a class may be in different behaviour phases, depending on the faulty behaviour exhibited by the external entity, i.e., each object characterised by a current behaviour phase[3].

In our approach, the abstract interface of an object will remain the same throughout its lifetime; however the responses to the messages received by an object can change depending on the object's current behaviour phase. So an object may go through different behaviour phases (e.g., normal and abnormal), but its abstract interface will be the same, independent of the current behaviour phase that the object is in.

Hence, it is desirable to find means by which objects representing such real-world entities can make behaviour changes such that:

(i) all objects of a given type have the same abstract interface, but individual objects may exhibit different behaviour phases depending on their current phase of behaviour, and

(ii) the phase changes are represented very explicitly rather than implicitly in the software, that is, instead of using status-checking code which is normally embedded in the implementation of various operations of a class, we choose

[3]Note that the type of an object is fixed, that is, it does not change during its lifetime. However, an object can traverse different behaviour phases, exhibiting different kinds of behaviour during its lifetime depending on its current phase, but its interface with the system will remain the same.

to represent the different behaviour phases explicitly in the design as a means of producing a better software structuring.

An overview of our proposed solution consists of defining a set of different subclasses related to a single real-world entity, each subclass implementing a different behaviour phase that the external entity can possess. These different forms of behaviour are defined statically for anticipated environmental faults, and runtime access to the current behaviour phase is implemented using *delegation*. In other words, it is arranged that each object representing a real-world entity "forwards" the execution of its operations to appropriate instances of these subclasses, depending on the object's current behaviour phase.

In this thesis we aimed, and indeed found it practicable, to avoid creating a new object-oriented programming language to solve our problems. Instead we have chosen, on grounds of practicality to base our experimental work on C++[150]. To demonstrate the feasibility of our proposed approach for dealing with environmental faults, we have designed and implemented a software train controller which is explained in detail in Chapter 4.

The remainder of this thesis is organised as follows. Chapter 2 provides background information and terminology related to object-oriented programming. Chapter 3 describes the main contribution of this thesis, an object-oriented approach for dealing with environmental faults. It also includes a description of some directly relevant work we have done, in collaboration with Robert Stroud, an object-oriented approach to dealing with software design faults, plus a briefly description of work by others related to hardware faults based on object-oriented techniques. Chapter 4 presents the design and implementation of a complex control application for a large model railway using the approach described in Chapter 3. Chapter 5 discusses some additional examples drawn from the object-oriented literature that are related to our approach, and finally, Chapter 6 draws some conclusions about the work presented and suggests future directions.

BLANK PAGE
IN
ORIGINAL

Chapter 2

Object-Oriented Concepts

> "... There is a great difference between knowing something and doing it well. Writing object-oriented programs is no different. It is not enough to know the basic constructs and to be able to assemble them together into programs. The experienced programmer follows principles to make readable programs that live beyond the immediate need."[131]

This chapter defines and discusses the fundamental concepts of object-oriented programming used throughout this thesis. Terminology has yet to become standardised in the object-oriented community. To avoid confusion and ambiguity within this thesis, it is therefore necessary to provide careful definitions of the basic concepts of the object model, as well as of a number of much newer and more controversial ideas.

The structure of this chapter is organised as follows. Section 2.1 reviews the basic ideas related to software modelling and design, highlighting the prime importance of abstraction, conceptual modelling and aspects of representation in the process of program design. Section 2.2 discusses the basic concepts of the object-oriented model. Section 2.3 introduces object-oriented methodologies for object-oriented design. Section 2.4 presents a survey of exception handling in object-oriented languages. Following this, an object-oriented approach for the structuring of complex applications that deals with faults is briefly introduced; this approach is explained in detail in Chapter 3. Finally, we make some concluding remarks.

2.1 Managing Complexity

> "All systems are infinitely complex. (The illusion of simplicity comes from focusing attention on one or a few variables.)."[54]

Real-world computer systems are intrinsically complicated, and they have become even more complicated since the requirements of the new applications have become more demanding: high size and complexity, high reliability and high performance. It is sometimes possible, even desirable, to ignore the complexity of computer systems, but doing so does not make the complexity go away because it will certainly appear somewhere else. Software applications that deal with complicated problems must cope with this complexity somewhere.

As humans, we employ many mechanisms for managing complexity, such as, abstraction, generalisation and aggregation. Abstraction is a means of avoiding unwanted complexity, and it is the most powerful tool that we have for dealing with our complex world.

2.1.1 Abstraction

One of the most important tasks in software development is the analysis of the problem domain and the modelling of entities and phenomena relevant to the application. This conceptual modelling involves two main aspects: *abstraction*, related to our thoughts in observing the domain and defining the objects, properties and actions, and *representation*, which is related to the notation adopted to express the concrete model ("the physical reality"). Therefore success in programming depends on designing a representation and a set of operations that are both correct and effective[159].

An abstraction describes the essential characteristics of an object that distinguish it from all other kinds of objects, and thus provides defined conceptual boundaries, relative to the perspective of the viewer. Abstraction, then, removes certain distinctions so that we can see commonalities between objects.

Three different important concepts can be abstracted from an observation: a category or class, an action and an attribute. Based on these notions, three basic operations can be identified which involve abstraction, each one with its corresponding inverse operation: classification/instantiation, generalisation/specialisation and aggregation/decomposition, which are discussed in turn now.

Classification. It describes a number of objects that are considered in a category together because they have similar characteristics. For example, two people John and Mary can be classified as *instances* of *Person.* Classification can be described as an *is-a* association. For instance, John *is-a Person.*

Generalisation. Another operation comes out when we observe, for example, two categories and abstract another category which is more general than the others. Generalisation enable us to say that all instances of a specific category are also instances of a more encompassing category, but not necessarily the other way around. Generalisation can be described as a *kind-of* association. For example, *Graduate_Student* and *Postgraduate_Student* can be considered special cases of *Student.* On the other hand, *Student* is considered to be a generalisation of *Graduate_Student* and *Postgraduate_Student.*

Aggregation. It is an operation whereby we identify that instances of a category are composed of instances of other categories (also referred to as composition). That is, aggregation is a mechanism for forming a whole from component parts. An aggregation can configure assembled structures, for example, a *Car* consists of its *Bodywork*, four instances of *Seat*, five instances of *Wheel* and its *Engine.* It reduces complexity by treating many object as one object. Aggregation can be described as a *consists-of* association.

The Importance of a Proper Classification

In object-oriented design, the recognition of the similarities among things allows us to expose the commonality within key abstractions, and eventually leads us to simpler designs. Unfortunately, there is no golden path to classification. The identification of classes and objects is not straightforward in the design of a system. When we classify, we try to group things that have a common structure or exhibit common behaviour. A prime example of classification is provided by the field of biology. Detailed classifications of a range of biological phenomena have been worked out over a period of time. For example, the term *carnivores* is commonly used to represent animals which share certain behaviour or possess certain characteristics, i.e., animal which eat meat.

The problem of classification has been the concern of many philosophers, linguistics, and mathematicians. Historically, there have only been three general approaches to classification[20]:

- *Classical Categorisation.* In the classical approach to categorisation, all the entities that have a given property or collection of properties in common

form a category. For instance, married people form a category: one is either married or not, and the value of this property is sufficient to decide to which group a particular person belongs. Classical category was first studied by Plato, and then by Aristotle and many other scientists.

- *Conceptual Clustering.* Conceptual clustering is a more modern variation of the classical approach, and largely derives from attempts to explain how knowledge is represented. In this approach, classes (clustering of entities) are generated by first formulating conceptual descriptions of these classes and then classifying the entities according to the descriptions.

- *Prototype Theory.* Prototype theory is the more recent approach to classification: a class of objects is represented by a prototypical object and an object is considered to be member of this class if only if it resembles this prototype in significant ways.

These three approaches to classification have direct application in the identification of classes and objects, and provide the theoretical foundation of object-oriented analysis and design. For instance, we could first identify classes and objects according to the properties relevant to our problem domain. If it fails to produce a satisfactory class structure, we can consider clustering objects by concepts. If it also fails, we can consider classification by association, through which classes of objects are defined according to how closely each resembles some prototypical object.

Object-oriented systems generally exhibit one or more techniques to support classification of objects with like behaviour. The most significant ways are:

- *Sets.* They are the most general way of representing classifications in a system in the sense that the result of any predicate is a set of objects fulling that predicate. The advantage of using sets is that there is a well-understood semantics from the field of mathematics. It is not common to find sets in object-oriented languages, but in object-oriented databases they are found with more frequency.

- *Abstract Data Types.* Types are essentially a form of classification. However, they are more specific than sets in the sense that they are purely concerned with the external interface of the object.

- *Classes.* It is a template which fully defines the behaviour of a group of objects in terms of the operations, representations and algorithms.

Abstraction Hierarchies

When a system has too many relevant details for a single abstraction to suffice, it can be decomposed into a hierarchy of abstractions. A hierarchy, then, can be defined as a ranking or ordering of abstractions, which allows relevant details to be introduced in a controlled manner.

There are two kinds of abstraction hierarchies that are fundamentally important in modelling and design: aggregation hierarchies, which turn a relationship between concepts into an aggregate hierarchy[18, 121]; and generalization hierarchies, which turn a set of concepts into a generic hierarchy[120]. With generalisation, we can build hierarchies of concepts, forming more and more general concepts. Aggregation can define hierarchies of part-whole configurations since parts can have their own components parts.

Generalization is one of the most important mechanisms we have for conceptualising the real world. Interestingly, database research had been almost exclusively concerned with aggregation (Codd's normal forms), while the area of knowledge representation in artificial intelligence had been principally concerned with generalization (semantic networks and frames). In programming languages, aggregation is related to "record structure" and generalization is related to "record variant structure".

We can observe that aggregation and generalization are independent notions. In other words, they are orthogonal concepts and when they are employed together one can obtain a rich tool for modelling.

2.1.2 Representation

The commonest abstract representations in computing are programming languages. Many computing scientists recognise the close interaction between the thought and the human language that expresses these thoughts. In fact, the nature of the language actually shapes and models our thoughts, and vice-versa: language and thought model mutually each other, there is no precedence between them[72]. The relation between a design language and programs is described by Wulf (as pointed out by Ghezzi and Jazayeri in [59]):

> The nature of language actually shapes and models the way we think... If, by providing appropriate language constructs we can improve the programs written using these structures, the entire field will benefit... A language design should **at least** provide facilities which allow

comprehensible expression of algorithms; **at best** a language suggests better forms of expression.

However, it is difficult to transform world abstractions into language abstractions directly, without intermediate steps. In this case, intermediate graphic notations are very useful to help the programmer in the representation of his/her abstractions. So we can define software development as a process by which we refine successive transformations from a high level representation to a low level representation executable in a computer. As a consequence, the whole process is dependent on the nature of the abstraction facilities provided by the programming language in use. If the language restricts the way that abstractions can be defined, it will constrain application modelling.

2.1.3 Evolution of Abstraction in Programming Languages

At the beginning of programming language development, assembly languages simply enabled designers to write programs based on machine instructions (operators) which manipulate the contents of memory locations (operands). Therefore, the level of data and control abstractions were very low. A great step forward occurred when the first major imperative programming languages - Fortran and Cobol - appeared. Fortran was important because it introduced the notion of subprograms units: functions and subroutines while Cobol introduced the concept of data description.

Subsequently, Algol-60 introduced the concept of block structure, procedure, etc. It influenced numerous successor languages so strongly that they are collectively called *Algol-like* languages. The design philosophy of Algol-68 was to choose an adequate set of concepts and to combine them systematically. But it was Pascal that turned out to be the most popular Algol-like language because it is simple, systematic, and efficiently implementable. Both languages were among the first with rich control structures, data types and type definitions, following the ideas of structured programming created by Wirth, Dijkstra and Hoare[39].

Since the 1970s language design has focused more on supporting programming in the large. This concerns the construction of large programs from modules. A module is any named program unit that can be implemented as a (more or less) independent component. A well-designed module has a single purpose and presents a narrow interface to other modules. Such a module is likely to be reusable, i.e., able to be usefully incorporated in many programs, and modifiable, i.e., able to

be revised without forcing major changes to other modules. Parnas[124] around 1970 advocated the discipline of *information hiding* (also known as encapsulation). The idea was to encapsulate each global variable in a module with a group of operations that alone have direct access to the variable. Other modules can access the variable only indirectly, by calling these operations.

Only the "what" is of concern to the user of the module. The "how" is of concern only to the implementator of the module. A module is said to encapsulate its components. To achieve a "narrow" interface to other modules, a module typically makes only a few components visible outside. Such components are said to be exported by the module. There may be other components that remain "hidden" inside the module. So encapsulation suggests that a data structure must be resident within a module. An interface provides the access to that data structure which is needed by other modules. Thus, communication among modules should be done through well-defined interfaces which prevent direct access to data structures inside a module. Encapsulation minimises inter-dependencies among separately written modules by defining strict interface.

There are at least two known language mechanisms that support the notion of modularity: *modules* and *abstract data types.* Languages like Modula-2 and Ada implement variants of these both mechanisms. Roughly, a *module* consists of two related parts: *module specification* (called the *spec*) and *module implementation* (called the *body*). The module spec is a set of declarations of data structures and procedure signatures. The module body essentially contains the implementations of the entities declared in the module spec. All entities found in the module body are private, that is, not visible to clients to the module spec. Each element declared in the module spec must have its implementation in the module body. However, the module body may contain additional data structures and procedures which are used to implement the visible entities declared in the module spec.

The notion of abstract data types is one of the most important ideas that emerged from research in programming languages. The term *abstract data type* refers to a concept in which data structures, and related operations which manipulate those data structures, are encapsulated within a protected region. A language is said to support abstract data types when it allows designers to define new abstract data types consisting of declarations that bring together operations which manipulate private data structures. However, languages such as Modula-2 and Ada, which have abstract data types with uniform external interfaces still have some limitations:

(i) a flat type system, i.e., a system developed as a collection of abstract data types has a flat, uni-dimensional structure: relationships between types

are either lost or partially registered, such as generalisation/specialisation hierarchies.

(ii) abstract data types "vanish" at runtime, that is, abstract data types do not constitute an adequate abstraction at runtime. Although abstract data types are very useful at the design and implementation phases, they disappear at runtime. As a consequence, the system turns out to be again an unstructured bunch of code lines grouped into modules.

(iii) abstract data types are limited in reducing the semantic gap between problem domain and the solution domain. For instance, if one considers a group of active, autonomous entities composing a universe intrinsically concurrent, abstract data types would not model this sort of situation well.

The next evolution step was the introduction of the object concept, first introduced in the late's 60 by Simula[38], and consolidated by Smalltalk[62] in the late's 70. Simula introduced the concept of objects, classes and inheritance. Essentially, the traditional model of object-oriented programming employs *classes* to describe objects. Classes contain the definition of objects' behaviour, thus providing typing capability. Existing classes provide default behaviour for new classes. This *inheritance* of behaviour results in code sharing. Essentially, object-oriented programming treats functions and data as indivisible aspects of objects in the problem domain.

The object-oriented paradigm has been influenced by the notion of abstract data types because an object can be viewed as an instance of an abstract data type, which encapsulates a data type and provides a defined set of operations to manipulate and access that data type. Actually, in most object-oriented languages, a class definition describes a data type and the operations which can be performed on that data type. Furthermore, the concept of an abstract data type assumes an important role within an object-oriented approach because it may be seen as a way of providing abstract and simplified representation for a software system. In addition, abstract data types bring other benefits such as modularisation and encapsulation that are also relevant to the object-oriented paradigm.

2.1.4 Program Structuring Concepts

Computer scientists have recognised some time ago that mastering complexity was the key to successful software development. The concept of higher-level languages and their compilers was a large step toward this goal because it allowed

the programmer to work without being an expert on the details of the machine. After that, people began to recognise the need for better methods and tools to manage complexity; as a consequence, ideas such as structured programming and program development libraries, appeared.

Although these contributions have been valuable, they still leave a lot to be desired. In other words, there is much more to the subject of complexity than a simply attempting to minimise the local complexity of each part of a program. A much more important type of complexity is global or structural complexity: the complexity of the overall structure of a program or system (that is, the degree of association or interdependence among the major pieces of a program).

Complexity is one of the major causes of unreliable software, and it is both difficult to define precisely and to quantify. However, we can say that the complexity of an object is some measure of the mental effort required to understand it. In general, the complexity of an object is a function of the relationships among the components of the object.

The major visible difference between good and poor program structures seems to be complexity. Some concepts can be adapted from general systems theory to reduce complexity in software, such as[115]:

(i) partitioning the system into parts having identifiable and understandable boundaries,

(ii) representing the system as a hierarchy, and

(iii) maximising the independence among the parts of the system.

The act of partitioning a program into individual components can reduce its complexity to some degree. Although partitioning a program is helpful, a more powerful justification for partitioning a program is that it creates a number of well-defined interfaces within the program. These interfaces are invaluable for the understanding of the program. The interface show us which items are relevant and which are not, narrowing the focus of attention. In other words, the interface hides "irrelevant" information behind it.

The concept of hierarchical structure is of vital importance in both understanding and constructing systems. Because the average human mind has a rather small upper limit on the number of facts with which it can deal simultaneously, we find we can understand systems better if they are hierarchically defined[139]. Hierarchies allow the stratification of a system into various levels of detail. Each

level represents a set of aggregate relationships among parts in the lower levels. The concept of levels allows one to understand the system by hiding unnecessary levels of detail.

Although partitioning and hierarchical structure are important concepts in system structuring, there is a third related concept that is most important: independence. This concept implies that, to minimise complexity, the independence of each component in the system must be maximised. So the problem is not simply partitioning a program into a hierarchy, but determining how to partition a program into a hierarchical structure such that each module is as independent from all other modules as possible. Thus, *modularity* can be defined as being the property of a system that has been decomposed into a set of cohesive and loosely coupled modules.

Another important concept related to system structuring is making connections between components apparent as much as possible[114]. Usually in large systems is common to find many side-effects which make the system difficult to understand.

2.2 Object-Oriented Programming

Object-oriented programming is a model of programming based on concepts such as object, type, implementation hiding, inheritance and parameterisation. Object-oriented design is a way of using these concepts to structure and construct systems. Object-oriented programming and object-oriented design are intrinsically language independent. Consequently, in this section, the fundamental object-oriented concepts are defined independent of language concerns.

The essence of object-oriented programming is that of solving problems by identifying the real-world objects in the problem domain and the processing required by those objects, and, then, creating simulations of them. This idea of "programs simulating real world" grew from Simula 67, which was designed for actual simulation applications. Because much of the real world is populated by objects, a simulation of a such world should include simulated objects, which can send and receive messages, and react to the messages they receive.

As a consequence, object-oriented programming implies that the programmer views his/her program execution as involving objects which communicate using messages. Thus programming is achieved by class definitions and hierarchy creation, in which common properties are transmitted from superclasses to sub-

classes by the concept of inheritance. This style of programming has the following positive characteristics: modularity, support for generalization/specialization, a balancing view between data and process (an object can be either data or process), evolutionary and incremental activity, and reusability.

Much of the interest in object-oriented programming and object-oriented design is due to the growing industrial perception that it is a better way of structuring and building complex systems than other methodologies, since it promotes the reuse of software which improves the quality and reduces the cost of software development.

As mentioned at the outset of this chapter, terminology and even to a degree concepts have yet to become standardised in the object-oriented community. For this reason, the fundamental terms of the object-oriented model, which are used throughout the thesis, are carefully defined below. The definitions given here may differ wholly or partially from those used by other authors. However, efforts were made to bring some consensus among the great deal of different definitions, different jargons and different notations used by the object-oriented community.

Peter Wegner has proposed that we distinguish between object-based and object-oriented languages[156]. A programming language is called *object-based* if it is based on objects. A language is called *object-oriented* if it provides linguistic support for objects and additionally requires that objects are instances of classes. Furthermore, an inheritance mechanism must be supported. Thus:

$$\boxed{\text{OO Programming} = \text{Objects} + \text{Classes} + \text{Inheritance}}$$

According to this classification, languages, such as Ada and CLU, are object-based while Smalltalk and C++ are object-oriented languages. A more encompassing classification is proposed by Blair *et al.* which also includes delegation-based languages[16]. Many surveys of object-oriented languages can be found in the literature[133, 24, 102].

The object model comprises a collection of principles which forms the foundation of the object-oriented paradigm. The next subsections cover the concepts, features and mechanisms which are basic to the object-oriented paradigm, and set the terminology to be used in the reminder of this thesis. A key issue of object-oriented programming is to obtain reusable software components. This is achieved through the notions of object, class, inheritance, dynamic binding and polymorphism. The concepts of inheritance and dynamic binding are specific to object-oriented programming. In the following, we gradually introduce these concepts, starting with object and class, and then extend with inheritance, dynamic

binding and polymorphism. Finally, after the introduction of the basic concepts and terminology, we will take a closer look at object-oriented systems and will describe some variations that exists among the various implementations of the object model.

2.2.1 Data Abstraction

The principle of *data abstraction* emerged in the 1970's as a major technique in mastering complexity. Data abstraction is concerned with providing and abstraction over data structures in terms of a well defined interface. The advantages of data abstraction are well known, namely:

(i) the fact that the code and data structures concerned with a particular abstraction is recorded in a single place leads to well structured, understandable code which can easily be modified, and

(ii) the information hiding aspect provides a level of protection against unexpected access to data structures maintaining the object data integrity.

These same advantages are also claimed for object-oriented languages; clearly there are strong similarities between data abstraction and the concept of class. Moreover, there has recently been great interest in the semantics of typing in an object-oriented context, in particular, concerning the integration of static type checking into object-oriented systems[64, 99]. In this section, the important concepts of object, class and type are discussed. Abstract data types, which have strongly influenced the object-oriented paradigm, are highlighted as a central feature of statically typed object-oriented languages.

2.2.1.1 Objects

Objects are entities that encapsulate state information or data, and a set of associated operations that manipulate the data. In general, each object's state is completely protected and hidden from other objects, and the only way of examining it is by making an operation invocation on one of the object's publicly accessible operations. Objects have a unique identity. *Identity* is the property of an object which distinguishes it from all other objects.

2.2.1.2 Classes

A *class* is a template description which specifies properties and behaviour for a set of similar objects. Every object is an instance of only one class. Every class has a name and a body that defines the set of attributes and operations possessed by its instances. Note that the term object is sometimes used to refer to both class and instance (especially with languages like Smalltalk where a class is itself an object). However, it is important to distinguish between and object and its class; here the term class is used to identify a group of objects and the term object to mean an instance of a class.

In object-oriented programming, attributes and operations are usually part of the definition of classes. Attributes are named properties of an object and hold abstract states of each object. Operations characterise the behaviour of an object, and are the only means for accessing, manipulating and modifying the attributes of an object. An object communicates with another through messages which identify operations to be performed on the second object. The object responds to a message by possibly changing its attributes or by returning a result. The interface comprises of the set of operations which can be requested by other objects; the external view of an object is nothing more than its interface. In object-oriented programming languages, operations that client may perform upon an object are typically declared as *methods*, which are part of the declaration of the class of the object. C++[94] uses the term *member function* to denote the same concept, and here we will use the terms *method* and *member function* interchangeably. Also, for our purposes, a message is simply an operation that an object performs upon another, so the terms *message* and *operation* are interchangeable.

Some important software engineering principles such as data abstraction, encapsulation and modularity are achieved with the use of concepts involving classes and objects as stated above. These characteristics are widely recognised as being good qualities of a software system, therefore the object model, in principle, encourages high quality software development.

2.2.1.3 Types

Following Ghezzi and Jazayeri[60], the concept of *type* can be viewed as a specification of the set of values that can be associated with a variable, together with the operations that can be legally used to create, access, and modify such values. We refer to the set of operations as the public interface of the type. For example, type *boolean* is bound to a certain class of values *true* and *false* and to the

operations **and**, **or** and **not**. In an object-oriented context, one could give another definition of type in terms of objects. *Type* can be defined as the collection of all objects in a system that respond in the same way to the same set of messages. In other words, a type is a collection of objects with the same public interface[11].

The concepts of type and class are distinct[17]. Type is essentially the description of an interface. This interface specifies a behaviour that is common to all objects of a given type. A *class* specifies a particular implementation of a type. A class definition is common to all instances of the class which includes a description of the internal state of the object and its methods. Every object is an instance of some class. Thus the type of an object depends on its interface with the "outside world" rather than the class from which it was instantiated.

Although the terms type and class as described here are not the same thing, some authors use the terms interchangeably[20]; others state that the distinction between the two concepts is primordial[125]. More recently some object-oriented languages, such as Arche[15], Fibonacci[3], POOL-I[7] and Guide[85], cleanly separate these two concepts. These languages explicitly separate the definition of type and its implementation. Some authors argue that this separation is confusing[20], and they believe to be sufficient to say that a class implements a type. In fact, none of the widely used object-oriented languages, such as C++ and Eiffel, currently provide this separation.

Types can be bound to variables either statically or dynamically[1]. *Static typing* is when the type of a variable referring to an object is constrained before runtime. At runtime, the object can in fact belong to a subtype of the declared type. The validity of the operations is checked at compile time. Languages such as C++, Eiffel and Simula are statically type-checked. A language is said to be *strongly* typed if it allows all type checking to be done statically.

Dynamic typing is when the type of a variable referring to an object is known and can dynamically be changed at runtime. As a consequence, the validity of the operations can only be checked at runtime. Smalltalk and Objective-C are examples of dynamically typed languages. There has recently been great interest in the semantics of typing in the object-oriented context in order to check the correctness of the source code, especially with regard to messages and accesses to instance variables. In particular, many investigations concern the integration of static type checking into object-oriented systems[64, 99].

[1] Following the Eiffel model, we also make a distinction between objects and variables which refer to (are attached to) objects. Variables hold references to objects; they are not objects.

2.2.1.4 Abstract Data Types

The previous paragraphs highlighted the important distinction between class and type, which is a consequence of the separation of concerns between the specification of behaviour and the implementation of behaviour. In providing statically typed object-oriented languages, it is therefore almost inevitable that this separation will be a feature of the language design.

Abstract data types extend the principle of data abstraction by separating the specification of data abstraction from its implementation. This was an important improvement because the abstraction gained from abstract data types is reflected directly in the syntax and in the semantics of the language. Abstract data types can be considered to consist of two parts (Figure 2.1): the *specification* part (i.e., the interface) and the *implementation* part. Each part can be further subdivided with the specification being denoted by the *syntax* (signature) of the specification together with the *semantics*, and the implementation part denoted by the *representation* (data structures) and the associated algorithms.

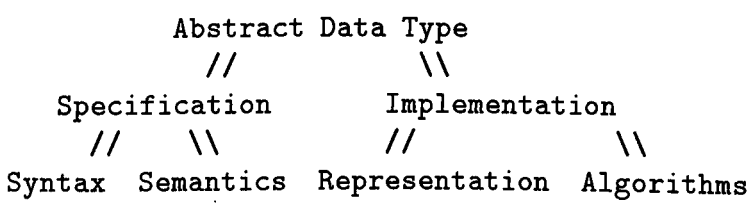


Figure 2.1: Structure of an Abstract Data Type

The benefit derived separating specification from implementation is that of allowing one to reason about and use a type without knowing anything about its implementation. As a consequence, a type can have one or more implementations. However, to demonstrate the correctness of the implementations, it is necessary a description of the type's behaviour that is implementation free.

The syntax of an abstract data type is normally referred to as the *signature* which defines the interface. However, such a syntactical specification is not sufficient to describe the behaviour of a data type. It is necessary to have some kind of support for specifying also the semantics of the data type. One of the most successful formal techniques for specifying the semantics of a data type is called *algebraic specification*[47]. This technique requires the programmer to define a set of equations related to the operations of the data type.

Another approach to help the programmers better express the behaviour of abstract data types relies on the provision of pre- and post-conditions on methods. *Predicates* (boolean expressions) can also be associated with the operations in order to describe the desired state of the object. A *precondition* for an operation is a predicate concerning the state of the object before the operation is invoked against it. If the precondition is true, the operation can be safely executed. If it is false, nothing can be said about the invocation. A *postcondition* is a predicate associated with the state of the object after the operation being executed.

This is the approach taken by Eiffel[105]. For example, in Eiffel a programmer can attach pre- and post-conditions to the Push and Pop operations to guarantee the semantics of a stack. A precondition for Push is the requirement that the stack instance must not be full. A precondition for Pop is the requirement that the stack instance must not be empty. Similarly, a postcondition for Push is the requirement the the stack instance is no longer empty, and for Pop is the total number of elements is increased by one.

It should be stressed that most programming languages do not make use of semantic specifications. On the contrary, many languages require only the specification of the syntax. The object model is still lacking a profound theoretical understanding, but some investigations have been appearing in this area. For instance, object-oriented specification[97, 98] is becoming an influential technique. Bar-David[11] describes the behaviour of a type through a set of equations that relate the public interface of the type in the context of C++.

2.2.1.5 Encapsulation

Following Snyder[141], *encapsulation* is defined as a technique for minimising interdependencies among separately-written modules by defining strict external interfaces. The external interface of a module serves as a contract between the module and its clients. A module is *encapsulated* if clients are restricted by the definition of the programming language to access the module only via its external interface. Thus encapsulation assures designers that compatible changes can be made safely, which facilitates program evolution and maintenance. These benefits are especially important for large and long-lived systems.

In a typical object-oriented language, a *class* definition is a module whose external interface consists of a set of operations; changes to the implementation of a class that preserve the external interface do not affect code outside the class definition. The class mechanism allows the programmer to encapsulate syntactically a description of the data structure together with the public interface.

However, inheritance introduces a new category of client for a class: its derived classes. (Snyder[141] provides an excellent discussion of encapsulation in the presence of inheritance.) So we should also consider what external interface is provided by the class to its derived classes. This external interface is just as important as the external interface provided to the users of the objects, as it serves as a contract between the class and its derived classes, and thus limits the degree to which the programmer can safely make changes to the class. In other words, a class has two kinds of clients[82]:

Instantiating Clients. Those which just create instances of the class and manipulate those instances through methods associated with the class.

Inheriting Clients. Those which are subclasses and inherit methods and structure from the class.

Object-oriented languages have differed in the strategies for giving access and visibility to these two kinds of clients. For example, Smalltalk allows unrestricted access to instance variables by the inheriting clients, but completely restricts instantiating clients. Numerous languages constructs have been proposed in order to distinguish the "public" features from the "private" features. It seems that the most general approach, which combines efficiency and flexibility, is to support the notions of public, private and subclass visible, and leave it to the programmer to specify the desired protection[82]. The three options are defined as follows:

Public. If an instance variable or a method is declared to be public, then any client can directly access, manipulate, or invoke it.

Private. If an instance variable or a method is declared to be private, then no client can directly access, manipulate, or invoke it.

Subclass Visible. If an instance variable or a method is declared to be subclass visible, then it can be accessed, manipulated, or invoked directly only by inheriting clients.

C++, for example, uses *class* declaration in conjunction with the access keywords *private*, *protected* (subclass visible) and *public* to provide encapsulation of the methods and instance variables.

Table 2.1 summarises how well some object-oriented languages support encapsulation to instantiating clients. It is clear that the language should support a way

	Visibility of Representation	*Visibility of Operations*
C++	data in public part can be accessed directly	only public visible
Trellis/Owl	can only be accessed through operations	only public visible
CommonObjects	can only be accessed through operations	all visible, but can use Common Lisp packages to export public ones
Simula	can be accessed directly	all visible
Smalltalk	can only be accessed through operations	can be marked private, but not enforced by the language

Table 2.1: Interface to Instantiating Clients

of separating the public interface from the private implementation details. An object should only be manipulated through a specific set of operations. Moreover, a language that enforces such information hiding minimises the effect of changes to a class on its clients.

Table 2.2 summarises the support for encapsulation to inheriting clients offered by some object-oriented languages. It is clear that a language should enforce that instance variables in ancestors should only be accessible through operations defined in the ancestor. Moreover, it is important for a language to support a way of making operations visible only to its inheriting clients, and not visible for casual, non-descendent clients.

2.2.2 Inheritance

This section discusses inheritance and how it supports hierarchy. Firstly, we begin to talk about inheritance, and then we discuss two major uses of inheritance, *implementation hierarchies* and *type hierarchies*[96, 141]. Following that, we discuss the use of multiple inheritance when a new class needs to inherit properties from two or more parent classes.

Inheritance (also called *subclassing*) is a mechanism for deriving new classes from existing classes by a process of refinement. A derived class inherits the data representation and operations of its parent class but may selectively add new opera-

	Access of Instance Variables	*Visibility of Methods*
C++	by name allowed	only public and protected visible
Trellis/Owl	by accessor function only	only public and subtype-visible
CommonObjects	by accessor function only	all visible
Simula	by name allowed	all visible
Smalltalk	by name allowed	all visible

Table 2.2: Interface to Inheriting Clients

tions, extend the data representation or override the implementation of existing operations[89]. According to Liskov[96], there are two major uses of inheritance, implementation hierarchy and type hierarchy, which are now discussed in turn.

2.2.2.1 Implementation Hierarchies

The first way that inheritance is used is simply as a technique for implementing abstract data types that are similar to other existing types. As an example of implementation inheritance, suppose that we want to implement a *Stack* class, and we have already implemented a *List* class (Figure 2.2). Then we can implement *Stack* as a subclass of *List*. Pushing an element on the top of the stack can be achieved by adding an element to the end of the list and popping an element from a stack corresponds to removing an element from the end of the list.

```
List      Public Interface: add
  |                         addEnd
  |                         remove
  |                         removeEnd
Stack     Public Interface: pop
                            push
```

Figure 2.2: An Example of Implementation Inheritance

In this case, the programmer uses inheritance as an implementation technique with no intention of guaranteeing that the subclass has the same behaviour as

the superclass. It just happens that an existing class already implements some behaviour that we want to provide in a newly-defined class, although in other aspects the two classes are completely different. This use of inheritance can lead to problems if other operations that are inherited provide unwanted behaviour. So inheritance of implementation is not recommended because it can lead to incorrect behaviour.

2.2.2.2 Type Hierarchies

A type hierarchy is composed of subtypes and supertypes. A type S is a *subtype* of T if and only if S provides at least the behaviour of T[17]. An object of type S can thus be used as if it is of type T because it is guaranteed to provide at least the operations of T. That is, an object of type S is *substitutable* for an object of the type T[96]. So the key is that derived classes behave like parent classes. This is also referred to as *conformance*, i.e., type S conforms to (is subtype of) type T.

This use of inheritance relates together the behaviour of two types, and not necessarily their *implementations*. A derived type may have additional properties and operations that have nothing to do with its parent type. Behaviour sharing via class inheritance is justifiable only when a true generalisation/specialisation relationship occurs, that is, only when it can be said that the subclass is a form of the superclass[96]. When a class B *behaviourally inherits* from A, we assume that every instance of class B is an instance of class A because it behaves the same. From now on, we refer to this model of inheritance as *behavioural inheritance*[11, 6].

We begin with some examples of types that are not subtypes of one another. First, a stack is not a subtype of a list nor the reverse is true. If stack is implemented as a subclass of list, we are also inheriting an unwanted list of operations that add and remove elements from arbitrary positions in the list (not only add and remove from the beginning). If these operations are used by mistake the stack will not behave as expected. Another example of non-subtypes are stacks and queues. Stacks are LIFO, i.e., when an element is removed from a stack, the last item added is removed.

By contrast, queues are FIFO, i.e., an element is added in the end of the queue and an element is removed from the beginning of the queue. As an example of a truly subtype hierarchy, consider an *indexed collection* which has an operation to access elements by index, where all subtypes have this operation too, but, in addition, each would provide extra operations. Examples of subtypes are *arrays*, *sequences*, and *indexed sets*. Figure 2.3 illustrates another example of a truly

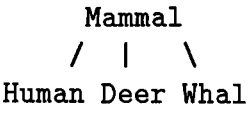


Figure 2.3: An example of Type Hierarchy

subtype hierarchy. Behavioural inheritance is often called the *isA* relationship, that is, it describes one kind of object that as an special case of another. For example, we could say that array is a special kind of indexed collection.

Behavioural Inheritance

The contradictory uses of inheritance, implementation inheritance and behavioural inheritance, have elicited suggestions to confine its use to the representation of generalisation/specialisation relationship, that is, to the use of behavioural inheritance[96, 11, 103, 119]. As mentioned earlier in 2.2.1, some object-oriented systems like POOL-I[7] and Guide[85] cleanly separate the concept of class (implementation) from type (specification). However, none of the mainstream object-oriented languages, such as C++ and Eiffel, currently provide this separation. For this reason, throughout this thesis, we adopt a design philosophy in which inheritance is used exclusively to model the specialisation relationship, that is, we adopt the behavioural inheritance model that is increasingly being accepted as the model that supports a subtyping/conformance relationship.

2.2.2.3 Multiple Inheritance

So far our examples have used *single inheritance*, that is, each subclass had one and only immediate superclass. The class inheritance hierarchy with single inheritance is a tree. In many situations, however, it is desirable to inherit from more than one class. This mechanism is termed *multiple inheritance*. With multiple inheritance, we can combine several existing classes to produce combination classes that utilise each of their multiple superclasses in a variety of functionalities. Then the class inheritance hierarchy becomes a DAG (directed acyclic graph), since a class can have more than one immediate predecessor. For example, a BorderedTextWindow that allows editing of text in a bordered window inherits from both TextWindow and BorderedWindow (Figure 2.4). Another example of multiple inheritance is extracted from [48]. Assume that we have three base classes Mammal, Bird and FlyingThing. They define the common behaviour for mammals, birds and things that fly, respectively. We can now inherit from

Bird and FlyingThing to define a flying bird, for example, a swallow. In the same way, we can inherit from Mammal and FlyingThing to define a Bat (Figure 2.5).

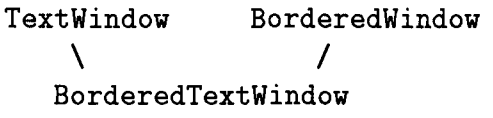


Figure 2.4: An Example of Multiple Inheritance

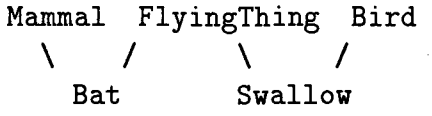


Figure 2.5: Another Example of Multiple Inheritance

When a class inherits from more than one parent there is possibility of conflict: methods or instance variables with the same name but different or unrelated semantics that are inherit from different superclasses. Many solutions have been reported in the literature to solve this problem. The following are some strategies for resolving conflicts of unrelated methods:

Linearisation. It specifies a linear, overall order of classes, and then specify that application of a class attribute (a method or instance variable) start from the most specific class. This is the approach taken in Flavors[113] and CommonLoops[19].

Renaming Strategy. It requires the renaming of conflicting instance variables or methods. This is the approach taken in Eiffel[105] and Trellis/Owl[134]. This approach is to issue an error if totally unrelated class attributes are inherited through the multiple inheritance mechanism. In Eiffel, for example, although such conflicts and ambiguities are forbidden, the language provides considerable flexibility for defining how the conflict is to be resolved.

Qualified Class Attributes. It requires that whenever there is an ambiguity in the access or usage of a class attribute, it must be resolved through qualifying the class attribute with the appropriate class name. C++[150] uses this strategy in supporting multiple inheritance.

Linearisation hides the conflict resolution problem from the user, but introduces a superfluous ordering of class inheritance semantics. The renaming and qualifying strategies provide more flexibility to the programmer to decide the applicability of an inherited method or instance variable. In terms of flexibility, generality and ease of understanding, these two approaches seem to be the most promising strategies.

A number of discussions of the viability of multiple inheritance have appeared in the literature[28, 140]. Multiple inheritance is a controversial issue; there has been a significant amount of discussion on the value of multiple inheritance over single inheritance. The precise semantics of single inheritance is still subject of many investigations, and multiple inheritance adds much more complexity to the subject; although some researchers, such as Cardelli[25], have already reported some studies on the definition of a clean semantics for multiple inheritance. Multiple inheritance can be useful in some situations for class specifications. The disadvantage of its use is, to some extent, a loss of conceptual and implementation simplicity. However, in my opinion, multiple inheritance is useful if applied with the appropriate assumptions. One should take care in defining a consistent and correct semantics when conflicting methods and instance variables from different classes are inherited by the same class.

2.2.3 Dynamic Binding and Polymorphism

This section examines in some detail the concept of dynamic binding and polymorphism, concentrating on the various styles of polymorphism in language and system design. Polymorphism and dynamic binding emerge as important concepts, especially in the context of object-oriented computing.

2.2.3.1 Dynamic Binding

In a general sense, *binding* is an association, possibly between an attribute and an entity. Examples of program entities are variables, subprograms, and statements. The attributes of a variable are its name, type, and storage area. The time at which a binding takes place is called *binding time*. Bindings and binding times are very important concepts in the semantics of programming languages. A binding is *static* if it occurs before runtime and remains unchanged throughout program execution. If it occurs during runtime or changes in the course of program execution, it is called *dynamic* or *late*. The major advantage of dynamic binding is that bindings can change over time, providing a great deal of programming flexibility.

Our main concern here is the binding of types to variables. Before a variable can be referenced or assigned in a program, it must be bound to a type. The two important aspects of this binding are how the type is specified and when the binding takes place. Types can be specified statically through some form of explicit or implicit declaration. Examples of typed object-oriented languages are Eiffel, C++, and Simula. All of them combine *static type checking* and *dynamic binding*. For example, assume that the variable *John* is of type Person, *Mary* is an Employee and *Jill* is a SalesPerson. Moreover, if Employee is a subtype of Person and SalesPerson is a subtype of Employee, the following assignments are valid (using syntax similar to C++):

```
John := Mary;
John := Jill;
Mary := Jill;
```

Note that, for example, the assignment *John := Mary* is dynamically binding *John* to an object of a different type (that is, different from its static type). In principle, it would be a violation of strong typing to assign objects of a different type to any of these variables. But, as long as Employee is a subtype of Person and SalesPerson is a subtype of Employee these assignments are valid.

In terms of object-oriented programming, binding is normally concerned with the mapping from method name to implementation, that is, it means that a message send at runtime is dynamically bound to an implementation depending on the class of the receiver. For example, assume that the method EvaluateBonus is defined in the class Employee and is *redefined* or *overridden* with an entirely different implementation in class SalesPerson. Then if EvaluateBonus is invoked on *Mary* before and after the assignment *Mary := Jill*, as in the following piece of program:

```
Mary.EvaluateBonus(...); // execution of Employee::EvaluateBonus
Mary := Jill;
Mary.EvaluateBonus(...); // execution of SalesPerson::EvaluateBonus
```

then the code defined in Employee is executed before the assignment, and the code defined in SalesPerson is executed after the assignment. So the programmer can have the flexibility of dynamic binding added to the advantages of strong typing. For example, in C++, one must explicitly request dynamic binding for a particular message by declaring it to be *virtual* in the parent class and redefining it in the derived class. However, operations of the subclass which override the

corresponding operation of the superclass have a responsibility to provide the same services provided by the superclass. This motivates our discussion of the next topic.

2.2.3.2 Design by Contract

In the object model, a derived class inherits the state and operations of its parent class, and may add new operations or modify the implementation of existing operations (*overriding of operations* as discussed above). Such modifications should be realised in such a manner so as to guarantee that the derived class still perform the same basic abstraction as the parent class. Meyer[107, 108] has proposed a contract model for inheritance, known as *design by contract*, in which the behaviour of a class is constrainted by preconditions and postconditions on its operations and a class invariant which must be satisfied by objects of the class.

This idea of "programming as contracting" is extended in the case of inheritance by the notion of *subcontracting*. Derived classes are obliged to conform to the specification of their parent classes in the sense that the invariant for the derived class must be at least strong as the invariant for the parent class. Moreover, for each operation of the derived class that overrides an operation of the parent class, the following assertion overriding rule should hold:

(i) preconditions may only be weakened in the derived class, and

(ii) postconditions may only be strengthened in the derived class, since the postcondition of the overridden operation in the parent class must still hold.

These two requirements produce an inheritance hierarchy in which the subclasses are subtype compatible with their parents and polymorphic substitution of instance of derived classes for instances of parent classes is possible without affecting the correctness of the overall system. So without assertions and the notion of contract, inheritance and overriding maybe misunderstood and misused.

The behavioural inheritance model earlier described should thus go beyond the simple textual matching of method names. It is important that overriding of operations by a derived class be performed in a disciplined manner so that the derived class still realises the same basic abstraction as the parent class. In other words, our concern is not only with syntax but also with semantics, more specifically with the preservation of the subtyping relationship by derived classes.

With such a view, in this thesis we adopt a model of inheritance that is termed *restricted inheritance*, which encompasses both notions of behavioural inheritance and "design by contract" methodology.

2.2.3.3 What Polymorphism Means

Typing systems in most programming languages are monomorphic in that values are considered to have a single type. Type checking is then performed on the basic of this typing information. However, in many cases, this can be too restrictive, and many languages designers started employing polymorphism as a means of providing more flexible typing disciplines. *Polymorphism* is defined to be the ability for some values and variables to have more than one type. So these values and variables can be used in different contexts demanding different type values. Polymorphic functions are functions whose operands (actual parameters) can have more than one type. Polymorphic types may be defined as types whose operations are applicable to operands of more than one type.

Polymorphism should be applicable to all types in a well designed language. In particular, given that functions are types, then a language should support polymorphic functions. A programming language has a polymorphic type discipline if it permits us to define a function which works uniformly for arguments of different types. For example, in a polymorphic language, we can define a single function **length** of type:

```
length::[A] -> num, for all types A

which means that
(i) length is a function whose arguments are lists,
(ii) its values are numbers, and
(iii) the type of the entries in the argument lists does not matter.
```

In contrast, a language with monomorphic type discipline forces the programmer to define different functions to return the length of a list of integers, a list of reals, and so on. Examples of languages with monomorphic types are Pascal and Algol68.

The above example illustrates the inflexibility in monomorphic languages. Although Algol60 presented a more flexible type discipline than Pascal in that it required procedure parameters to be specified only as "procedure" (rather than say "integer to real procedure), the flexibility was not uniform. A polymorphic

type discipline was first worked out for the language ML[110] around 1976, and since then has been incorporated in a number of functional and imperative languages.

Polymorphism was first identified by Strachey[145], and the topic is examined in detail in a paper by Cardelli and Wegner[26]. In this paper, Cardelli and Wegner give a classification that divides polymorphism into two categories called *ad hoc* and *universal* (Figure 2.6). Coercions and overloading are two forms of ad hoc polymorphism which are supported for many languages such as APL, Algol68 and Ada:

(i) *Coercions* provide a simple way of going around the rigidity of monomorphic languages and provide a limited form of polymorphism. Languages supporting coercion have built-in mappings (coercions) between types. If a particular context demand one type and a different type is provided, then the language checks if there is an appropriate coercion. For example, if **add** is defined on two reals and an integer and a real are provided as parameters then the integer will be coerced on to a real value.

(ii) *Overloading* allows a "function name" to be used more than once with different types of parameters. For instance, the **add** function could be overloaded to operate on both integers and reals as above. The typing information of the parameters will then be used to select the appropriate function.

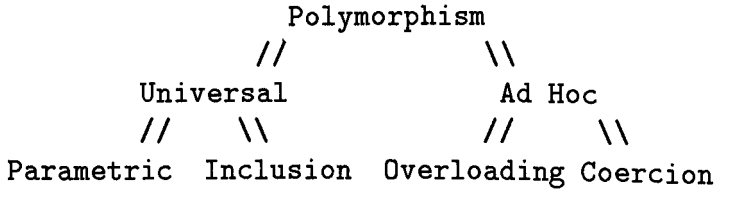


Figure 2.6: Cardelli and Wegner's Taxonomy

According to Strachey, in ad hoc polymorphism there is no single systematic way in which to determine the type of the result of a function from the type of its arguments. So ad hoc polymorphism works in a limited number of types in an unprincipled manner while universal polymorphism will work for a potentially infinite set of types in a principled manner.

Universal polymorphism has two forms: *parametric* and *inclusion*. In parametric polymorphism, a single function (coded once) will work uniformly on a range of

types. A function has an explicit or implicit type parameter that determines the type of the arguments for each application of that function. Parametric functions are also called *generic functions*, that is, the function works generically on a range of types. (This style of polymorphism is revisited in the Section 2.2.3.4). In inclusion polymorphism, an object may be viewed as belonging to many different classes that need not be disjoint, that is, there may be inclusion of classes. Parametric polymorphism describes the polymorphism found in ML and its derivates, whereas inclusion polymorphism is the style of polymorphism found in object-oriented languages such as Simula67 and C++.

According to Cardelli and Wegner[26], if typing is viewed as partially specifying the behaviour (or intended use) of associated values, then monomorphic type systems constrain objects to have just one behaviour, whereas polymorphic type systems allow values to be associated with more than one behaviour. Strictly monomorphic languages are too restrictive in their power of expression because they do not allow values, or even variables that denote values, to exhibit different behaviour in different contexts of usage. Languages like Pascal and Ada have ways of relaxing strict monomorphism including:

(i) *Overloading.* Integer constants may have both type: integer and real. Operators such as '+' are applicable to both integer and real arguments.

(ii) *Coercion.* An integer value can be used where a real is expected, and vice versa.

(iii) *Subtyping.* Elements of a subrange type also belong to superrange types.

As we discussed before, overloading and coercions are classified as ad hoc polymorphism. Subtyping is an instance of inclusion polymorphism. The idea of a type being a subtype of another type is useful not only for subranges of ordered types such as integers, but also for more complex structures. For example, a type representing Fiat cars, which is a subtype of a more general types such as Vehicle. Every object of a subtype can be used in a supertype context, in the sense that every Fiat car is a vehicle and can be operated on by all operations that are applicable to vehicles.

Parametric polymorphism describes the polymorphism found in ML and its derivates, whereas inclusion polymorphism is the style of polymorphism found in object-oriented languages such as Simula67 and C++. For instance, Simula's *classes* are user-defined types organised in a simple inclusion (or inheritance) hierarchy in

which every class has a unique immediate superclass. Simula's objects and procedures are polymorphic because an object of a subclass can appear wherever an object of one of its superclasses is required.

It is also possible to introduce other forms of polymorphism in object-oriented languages such as overloading and parametric polymorphism. It can actually be shown that most inheritance systems support overloading in conjunction with inclusion polymorphism[16]. In C++, for example, a function name or operator is overloadable, that is, a function is called based on its signature. Also, using *virtual* member functions in an inheritance hierarchy it allows a runtime selection of the appropriate member function. Such a function can have different implementations that are invoked by a runtime determination of the subtype.

Cardelli[26] has shown how polymorphism can be introduced into lambda calculus using universal and existential quantification. This encourages a pure view of polymorphism, and therefore, it is possible to study the semantics of polymorphism in isolation. For instance, universal quantification corresponds to parametric polymorphism and bounded universal quantification corresponds to subtyping (for further details in this topic, refer to Cardelli and Wegner[26]).

One advantage that a polymorphic language has over a monomorphic language is that it can be rigidly typed checked while retaining a great deal of flexibility. This feature is the key to understanding typed, object-oriented languages.

Polymorphic Type Checking and Binding

At first glance, it seems that there is a conflict between the flexibility of polymorphic language and the requirements for correctness through type checking. It is however possible to support a flexible interpretation of a piece of code and still guarantee absence of type errors.

For instance, consider the following piece of code:

```
   procedure X (parameter: some_type)
   Begin
     ...
   End;

Begin
  ...
  X(some_actual_type);

End.
```

In monomorphic languages, the interpretation of the "X(some_actual_type)" procedure call is simple. The procedure X can only take one possible parameter type as defined by the formal parameter. Type checking is therefore a case of checking the actual parameter against the formal parameter. It also can be deduced that there will be one to one mapping from "procedure name" to "code body". Thus binding is simply concerned with finding the corresponding code body.

However, in polymorphic languages, the relationship between type checking and binding becomes more complex. The procedure X may support a range of parameter types with each type possibly requiring a different interpretation of the procedure. It is this feature which introduces flexibility into polymorphic languages. Type checking guarantees that an interpretation exists for a given type. In contrast, binding resolves the exact interpretation for that type.

For instance, in our previous example in Section 2.2.3.1, *Mary* is an Employee, but it could denote objects of the type Employee and SalesPerson since SalesPerson is a subtype of Employee. Then depending on the type of the object held by *Mary* the implementation of the method EvaluateBonus is different.

Other Polymorphic Languages

There is whole range of polymorphic languages often based on parametric polymorphism such as ML[111],Miranda[153], Russel and Hope[21]. The big advantage of these languages is that interpretation of polymorphic code is resolved at runtime, that is, dynamic binding is used. This contrasts with Ada which also supports parametric polymorphism but interpretations are fixed at compile time. This approach corresponds to a macro expansion treatment of polymorphism, and it has some limitations because does not capture the true semantics of polymorphism. For instance, a polymorphic function is mapped to several monomorphic functions, one for each type. Then each monomorphic function is compiled and executed as a separate entity. In contrast, in a language like ML, there is only one code body for a function and the true interpretation of the code is resolved dynamically.

Finally, we should say that the concepts of inheritance and polymorphism provide the great strengths of object-oriented languages, but they also introduce difficulties in program analysis and understanding. Grogono and Bennett[63] assert that "although polymorphism is a powerful abstraction tool, indiscriminate use of polymorphism can quickly lead to unreadable code". Wilde and Huitt[158] analyse problems of dynamic binding, object dependencies, control of polymorphism, high-level understanding and detailed code understanding in object-oriented systems.

2.2.3.4 Genericity

Genericity or *parametric polymorphism* is the ability to parameterise a software element with one or more types. A *parameterised* or *generic* class is a class that serves as a template for other classes, in which the template may be parameterised by other classes, objects, and/or operations. A generic class must be instantiated (its parameters filled in) before objects can be created. The most classical example is the generic class Stack(T), where T is a parameter type. In this way, a generic stack program could be written independently of the type of the item in the stack.

Ada is a non-object-oriented language which supports genericity. More recently it has been implemented in several other languages such as C++[149] and Eiffel[105]. Eiffel distinguishes between constrained and unconstrained genericity. In constrained genericity, some specific requirements are imposed on generic parameters whereas unconstrained genericity has no such limitations. For example, assume we have a generic function which computes the minimum of two values:

```
function minimum (x,y: T) return T
begin
    if x <= y then return x
    else return y;
end;
```

Such a function declaration is only meaningful if instantiated for types T on which a comparison operator $\leq$ is defined. So it is necessary a way for specifying that type T must be equipped with the right operation. In Eiffel, the operator $\leq$ would be treated as a generic parameter of type T.

2.2.3.5 Abstract and Concrete Classes

Following Rumbaugh[131], an *abstract* class is a class that has no direct instances but whose descendent classes have direct instances. A *concrete* class is a class that is instantiable. An abstract class can define the protocol for an operation without supplying a corresponding method. This is called an *abstract operation.* An abstract operation defines the form of an operation for which each concrete subclass must provide its own implementation. For example, Figure 2.7 shows an abstract operation. Ring is an abstract operation of class Telephone; its form but not its implementation is defined. Each subclass must supply a method for the abstract operation. ISDNPhone and POTSPhone are examples of concrete classes.

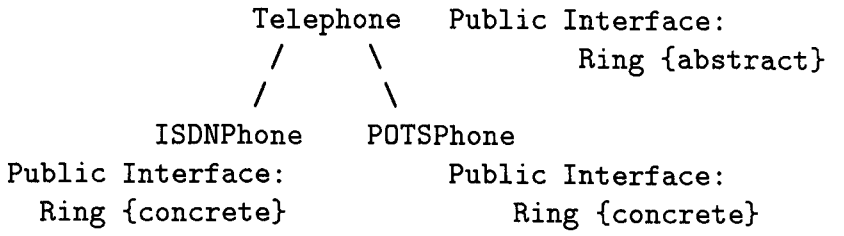


Figure 2.7: Abstract Class and Abstract Operation

Syntactically, an abstract class is expressed in Smalltalk simply by not specifying an implementation for at least one declared operation. In C++, an abstract class is created by declaring at least one operation as *pure virtual* (i.e., creating an abstract operation). In Eiffel, an abstract class is created by declaring the class and at least one operation as *deferred.* A summary of the above abstractness mechanisms is given in Table 2.3.

	Terminology Class	*Terminology Operation*	*Syntax Class*	*Syntax Operation*
C++	abstract	pure virtual	-	=0
Smalltalk	abstract	abstract	-	-
Eiffel	deferred	deferred	deferred with at least one deferred operation	deferred

Key Words:
'-' means that the feature is not applicable.

Table 2.3: Summary of Existing Abstractness Mechanisms

According to Hürsch[77], those definitions are problematic because there are two concepts involved in abstractness: ability to create instances, and presence of abstract operations. The fact that these two different concepts are merged into one single concept prevents the programmer from using them separately. If a class contains abstract operations then the class cannot be safely instantiated, because if an instance were created, it would not be able to successfully respond to all its messages. So the presence of an abstract operation implies the inability to instantiate objects. However, the converse is not true: a class might not be intended for instantiation and yet might have no abstract operations. So Hürsch proposes that the ability to instantiate objects should be the more important

behind abstractness, defining abstractness of a class as *the inability to instantiate objects.* Note that this definition deliberately does not make reference to the presence of abstract operations.

2.2.3.6 Covariance vs Contravariance

If a subclass inherits a method from a class which takes an argument of type T, the subclass can change the signature of the method to take any type that conforms to T. Languages that allow this, such as Eiffel, are said to have *covariant* typing[69, 70]. For instance, suppose that a class **Set** has the operation **add(object)**. A subclass **IntegerSet** which inherits from **Set** can modify the signature of **add(object)** creating a more restrictive operation **add(integer)**. Clearly, a **IntegerSet** does not want to invoke **add** on any kind of *object*, but only on the *integer* type. Thus, in **IntegerSet** we redefine the method **add** by changing the type of the argument.

According to Rumbaugh[131], overriding is done for the following reasons:

Overriding for Extension. The operation is the same as the inherited operation, except it adds some behaviour, usually affecting new attributes of the subclass. For instance, **Window** may have a **draw** operation that draws the window boundary and contents. **Window** may have a subclass **LabeledWindow** that overrides the **draw** operation. The **draw** operation in the subclass **LabeledWindow** can be implemented by invoking the method **draw** from the **Window** class and then adding code to draw the label.

Overriding for Restriction. The new operation restricts the protocol, such as tightening the types of arguments. This is exactly the case of covariant typing discussed above.

Overriding for Optimisation. An implementation can take advantage of the constraints imposed by a restriction to improve the code for an operation. For example, **IntegerSet** could have an operation to find the minimum integer. This operation could be implemented using a sequential search algorithm. A subclass **SortedIntegerSet** could provide a more efficient implementation of the operation since the contents of the set are already sorted. This is a valid use of overriding.

Overriding for Convenience. This is the practice of developing new classes using an implementation hierarchy, earlier discussed in Section 2.2.2.1. This practice is not recommended since operations might be overridden to make them to behave differently from inherited operations.

However, covariant typing presents some dangers. In general, clients expect to be able to call a method with any argument that conforms to the type T. If the subclass redefines the method to take another type T' (where T' is a descendent of T), and a client calls the operation with an argument that conforms to T but not T', the static type checking of the the system can be violated. Eiffel solves this problem by using a global type analysis, referred to as *system-validity checking* which is a more complex algorithm.

An alternative approach is to use *contravariant typing.* In this scheme, the original argument must conform to the redefined version. Object-oriented languages that only support contravariant typing, such as C++, eliminate the need for system validity checking. On the other hand, it constraints the programmer's ability to reuse code through inheritance. For instance, in C++, the signature of the operation **add(object)** cannot be changed in the subclass **IntegerSet**. This means that every time that the **add** operation is invoked on **IntegerSet** it is necessary to do a runtime check to ensure that the argument is really of the integer type, and not some other type of *object.*

2.2.4 Variations

Here we examine some variations in design choices and decisions related to the implementation of object-oriented systems. There are a number of different techniques that have been developed to support data abstraction and behaviour sharing. This section illustrates some of these variations by discussing concepts such as mixins, metaclasses, delegation, reflection and frameworks.

2.2.4.1 Mixins

Multiple inheritance from several distinct classes allows the inheritance of desirable and also undesirable functionality from these classes. It is not simple to selectively inherit only the desirable characteristics. The existence of multiple inheritance gives rise to a style of class derived from Flavors[113] called *mixin. Mixins* are packages of class operations that can be "mixed" in to other classes. Mixins are abstract classes in the sense they do not have any instances; they exist only to add functionality to existing classes. For example, suppose we have to define a class hierarchy representing plants, and we build two subclasses **FloweringPlant** and **Fruit/VegPlant**, both derived from the class **Plant**. However, we also need to model a plant that produces both flowers and fruits.

One solution is to create a third class **Flow/Fruit/VegPlant** which is derived from both **FloweringPlant** and **Fruit/VegPlant**. Suppose, by doing so, that the new class would contain duplicated information. So a better way is to use mixins. First, we create the classes **Fruit/VegMixin** and **FlowMixin** (Figure 2.8), which capture the properties unique to flowering plants, and fruits and vegetables, respectively. Neither of these classes can stand alone; rather, they are used to augment the meaning of some other class. Now, we can define **FloweringPlant** inheriting from **Plant** and **FlowMixin**. In a similar way, **Fruit/VegPlant** inherits from **Plant** and **Fruit/VegMixin**. Now we can declare the **Flow/Fruit/VegPlant** class inheriting from **Plant**, **FlowMixin** and **Fruit/VegMixin**.

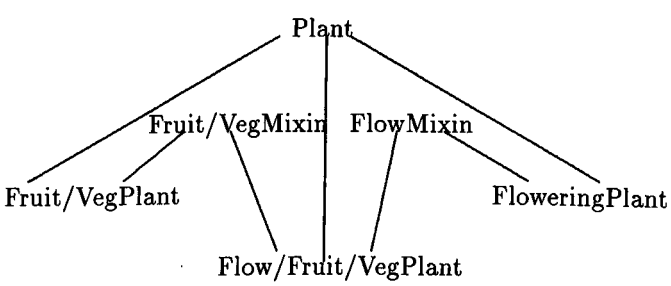


Figure 2.8: An Example of Mixins

In other words, a mixin is a class which embodies a single, focused behaviour. It is used to augment the behaviour of some other class via inheritance. Usually the behaviour of a mixin is completely orthogonal to the behaviour of the classes with which it is combined. The notion of mixin promotes reusability and separation concerns, if correctly applied. However, one problem with this approach is that if used unsystematically it can generate a large number of new classes for each original class. Every change in the functionality demands the creation of a new class before the instance can be created.

2.2.4.2 Metaclasses

In most object-oriented languages, classes are factories that create and initialise instances (objects). They are not first-class objects because they are not themselves instances of classes. However, in some languages such as Smalltalk[62], two types of generators are introduced: metaclasses, giving rise to classes, and classes, giving rise to terminal objects. Thus, all entities of the object model are objects.

More specifically, *metaclass* is a class whose instances are classes. There are at least two benefits in representing classes as objects. The first one is that the information global to all objects of a class can be stored in *class instances variables* (or simply *class variables*, following Smalltalk terminology). Methods associated with the class (called *class methods*) can be used to retrieve and update the values of class variables. The second advantage is its use in the creation/initialisation of new instances of the class. Each class can have, for example, its own overloaded *new* method for creating and initialising instances. Just as a class holds the description of its instances, so metaclasses hold the description of its single instance, i.e., the description of the class. There is (in Smalltalk, for example) a distinct metaclass for each class, and the metaclass hierarchy is parallel to the class inheritance hierarchy. Figure 2.9 illustrates this for the class hierarchy rooted at Person.

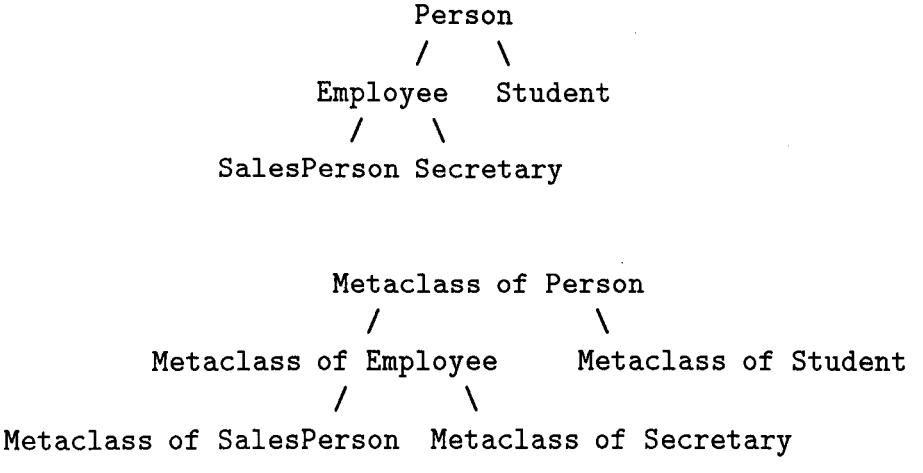


Figure 2.9: Parallel Class and Metaclass Hierarchies

Object-oriented languages such as C++, Eiffel and Simula do not support the "classes as instances of metaclasses" paradigm. Although instances of classes are also created through a *new* operator, the idea is to allocate dynamic memory rather than the creation of an instance as in Smalltalk. Some authors argue that the concept of metaclasses as implemented in Smalltalk is extremely complex[82], promoting the most significant barrier to learnability of the language by both students and teachers. However, in my opinion, the concept is useful although its understanding is not straightforward.

At its most general, the object-oriented paradigm supports three kinds of abstractions: (i) data abstraction for object communication, (ii) super-abstraction (inheritance) for behaviour sharing and object management, and (iii) meta-abstrac-

tion (metaclass) as basis for self-description. The concept of metaobject is also used to represent computational reflection in object-oriented systems. (Reflection is discussed further in 2.2.4.4.)

2.2.4.3 Delegation

The previous section discussed metaclasses. Object-oriented languages that support metaclasses have three categories of objects:

(i) Metaclass objects whose instances are classes.

(ii) Non-metaclass class objects whose instances are terminal objects.

(iii) Terminal objects that are not classes.

Inheritance applies only to classes. Now we discuss a mechanism like inheritance, but operates between objects directly, rather than between their classes, called delegation. In delegation-based programming languages, such as Self[154] and Actor[2], objects are viewed as *prototypes* (or *exemplars*) that delegate their behaviour to related objects, called *delegatees*. The *delegates-to* relationship can be established dynamically, whereas the inheritance relationship of class-based languages such as C++, is established and fixed when a class is created. So class-based languages use static and per-group behaviour sharing, in contrast to delegation-based languages which provide dynamic and per-object behaviour sharing.

In prototype systems, the distinctions between instance objects and class objects are removed. One first thinks of a particular prototypical object and then draws similarities and/or distinctions for other objects. Any object can become a prototype. The idea is to start with individual cases and then subsequently specialise or generalise them. In this sense, prototypes may be used in a manner close to the way humans learn. For example, assume that one observes a rectangular object, and then encounters a square. The square looks like the rectangle - it has four edges and four corners at 90 degree angles. Furthermore, suppose that subsequently one encounters another square of different size. Then one can deduce that the analogy with the first square is even stronger than that between the rectangle and the first square since the only difference between the two squares is their sizes. In a prototype system, the first rectangle is a prototype for the first square. Similarly the first square is a prototype for the second square.

Let us call the rectangle object **rec**, the first square **sq1** and the second square **sq2** (see Figure 2.10). The rectangle object has operations reflecting its behaviour, such as, **move**, **rotate** and **resize**, and also query operations[2], such as **center**, **perimeter** and **ratio**, which return the value of the attributes center, perimeter and ratio of the sides, respectively.

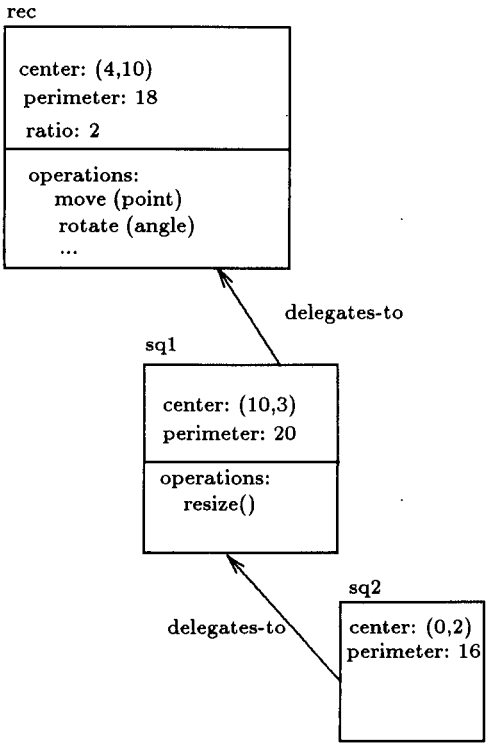


Figure 2.10: Rectangle and Squares Prototypes

The square object **sq1** is similar to the rectangle object; the only difference is in the values of center, perimeter and ratio of the sides (which for squares is 1). The **sq1** object has a delegates-to relationship with the **rec** object. This means state variables or operations that are not overridden in **sq1** will be inherited from **rec**. Thus **sq1** delegates the execution of messages, such as **rotate**, to the prototype object **rec**. In a similar way, **sq2** delegates all its received messages, except **Perimeter** and **Center**, to its prototype **sq2**. For example, if **sq2** receives the message **Move**, it will first be delegated to **sq1**. Since the message can not be handled by **sq1** it is in turn delegated to **rec**. Needless to say when **move** is executed it actually operates on **sq2**. In the next chapter we discuss more fully

[2]operations that read but not change attribute values.

the concept of delegation and its implementation in class-based object-oriented languages.

Inheritance vs. Delegation

To illustrate the difference between inheritance and delegation, consider a turtle-graphics example presented by Lieberman[93]. In this example, a turtle draws solid lines of length x when **forward** method is invoked. Backward movement is defined by the **backward** method, which invokes the **forward** method with a value $-x$. In a class-based language, the class **SolidTurtle** could look like this (here we use C++):

```
class SolidTurtle{
protected:
  changeDirection();
public:
  SolidTurtle();
  ~SolidTurtle();
  virtual void forward(int x); // draw solid lines of length x
  virtual void backward(int x);
}

void SolidTurtle::backward(int x)
{
  forward(-x);
}
```

The problem is to create a dashed turtle to draw dashed lines instead of solid ones. In a class-based language, a subclass **DashedTurtle** can inherit from **SolidTurtle**. The operation **forward** is then overridden in the **DashedTurtle** subclass, and the new implementation breaks the interval x into pieces, and invokes **SolidTurtle::forward** for each piece, thus drawing a dashed line. However, if a **backward** message is sent to a dashed turtle, the **backward** method of the **SolidTurtle** class is invoked. As a consequence, the **forward** method of the solid turtle is invoked instead of the specialised **forward** of the dashed turtle, thus incorrectly drawing a solid line.

One solution is to override the **backward** method in the subclass **DashedTurtle**, whose code would be exactly the same as **SolidTurtle::backward**. This solution is quite limited since the implementation of **DashedTurtle::backward** is a repetition of **SolidTurtle::backward**:

```
class DashedTurtle: public SolidTurtle{
public:
  DashedTurtle();
  ~DashedTurtle();
  void forward(int x); // break x into pieces and call Solid::forward
                       // for each piece
  void backward(int x);
}

void DashedTurtle::backward(int x)
{
  forward(-x);              // repeated code
}
```

Another solution is to change the implementation of the method **backward** in the **SolidTurtle** class. When a method in a superclass sends a message to *this* (the receiver of a message in C++ is called *this*), the method lookup starts in the class of the receiver. So **SolidTurtle** might have a **backward** method such as:

```
void SolidTurtle::backward(int x)
{
  this->forward(-x);
}
```

Sending the **backward** message to a dashed turtle will call the method defined in **SolidTurtle**. When **SolidTurtle** sends the **forward** message to itself, it calls the **forward** method that is defined in class **DashedTurtle**. Thus, the correct implementation of the **forward** method is invoked.

Now suppose that we prohibit the use of inheritance and we give to each dashed turtle an instance variable with a pointer to a **SolidTurtle**. Then a dashed turtle could respond to the **backward** message by sending it to **SolidTurtle**. However, the **SolidTurtle** would have to send the message **forward** back to the particular **DashedTurtle** that sent the **backward** message in the first place.

```
class DashedTurtle
{
private:
  SolidTurtle* solidTurtle;
public:
  DashedTurtle();
```

```
  ~DashedTurtle();
  void forward(int x); // break x into pieces and send the message
                       // 'forward' for each piece to the solidturtle object.
  void backward(int x); // send the message 'backward'
                        // to the solidTurtle object
}
```

However, this version of DashedTurtle is also incorrect, because without inheritance or late binding, it can only call the original forward method. The SolidTurtle would have to send the message forward back to the particular DashedTurtle that sent the backward message in order to invoke the specialised forward method. More generally, the SolidTurtle would have to send all messages overridden by subclasses back to the original receiver of the messages, which in our example is the DashedTurtle.

So delegation differs from just sending a message in that the delegator continues to play the role of receiver even after it delegates the message. Thus, messages that the delegatees sends to itself (that is, *this*) are received by the original delegator. Similarly, when a method in a superclass sends a message to *this* in a class-based system, the message lookup starts in the class of the original receiver. In this case, inheritance resembles delegation, while sending a message is different.

We implement delegation by including the original receiver as an extra argument to each delegated message. In our example, a pointer to a DashedTurtle is added as a parameter in the backward signature of the SolidTurtle:

```
void SolidTurtle::backward(DashedTurtle* dashedTurtle, int x)
{
  dashedTurtle->forward(-x); // pass control back to delegator
}
```

The result of this interaction enables DashedTurtle to delegate the backward message to SolidTurtle, which in turn returns *this* invocations to DashedTurtle, thus achieving the effect of late binding.

In a delegation-based system, the dashed turtle is created by cloning the solid turtle and adding a specialised forward method that will break the interval x into pieces and delegate each piece to the original solid turtle. If a backward message is sent to the dashed turtle, it is delegated to the original turtle. Then the solid turtle invokes the specialised forward method on the target turtle, thus drawing a dashed line (Figure 2.11). Languages based upon delegation, such

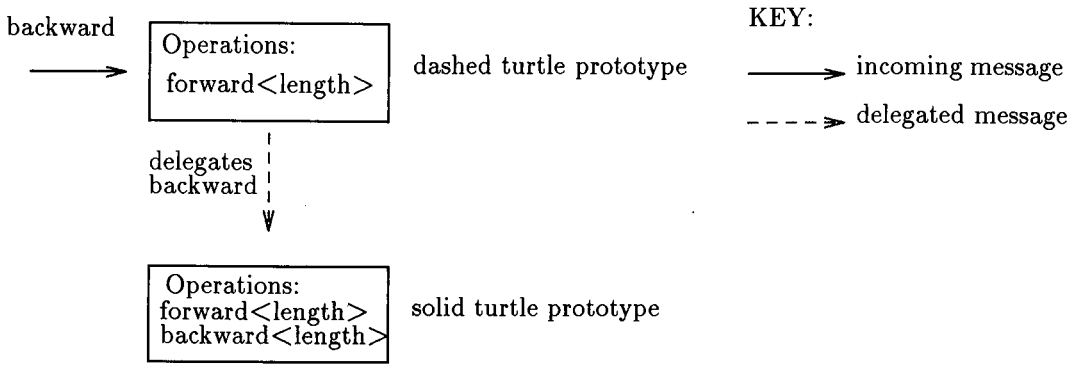


Figure 2.11: Example of Delegation

as, Self[154], implement the extra argument (that is, the original receiver of the message) automatically and invisibly. However, as we have shown above, it is possible to implement delegation in a class-based language by following a set of programming conventions.

2.2.4.4 Reflection

The notion of reflection originated in the field of formal logic, and concerns the ability of reasoning about and acting upon the system itself[101]. In the context of object-oriented programming languages, *reflection* can be defined as the ability of an executing system of programmed objects to make general object attributes, such as invocation, interface and inheritance, be themselves the subject of their computation[100]. The steps involved in reflection consist of: *reification*[3] of an abstract object-oriented concept, *reflective computation* using the reified attribute as data, and *reflective update* that modifies the objects through reflective computation.

More recently the prominence of reflection has been recognised and reflection has influenced the construction of adaptive, extensible systems, such as Apertos[162]. In the context of object-oriented programming languages, reflection has been combined with object-oriented techniques in the form of a *metaobject protocol*[83, 52, 50]. The idea is to associate an object with a *metaobject* which holds information about the object, such as the structure, behaviour, change history and other evolution information related to the object, and to provide in the language

[3]The Oxford dictionary defines reification as "the mental conversion of an abstract concept into a thing."

means by which the definition of these metaobjects can be changed. Any access to the object is intercepted by the metaobject, so that it can execute various tasks for the object. The Common Lisp Object System (CLOS) provides an example of such a metalevel facility[83]. As a result, by programming at the metalevel, it is possible to intercept and modify all the mechanisms by which objects and classes are defined. This is a very promising new technique, and investigations are being carried out within the Computing Science Department at Newcastle University in order to explore the use of reflection for implementing dependability and distribution transparency in software systems[148].

2.2.5 Summary

So far we have revisited the concepts of encapsulation, class, inheritance, etc., and also introduced the concepts of delegation, metaclasses, and reflection. Table 2.4 shows a comparison among some object-oriented programming languages according to the concepts discussed so far. Some object-oriented languages support the object-oriented paradigm better than others, although it is still possible to think in an object-oriented fashion without a direct support of the paradigm by the language. However, languages such as C++, Eiffel or Trellis/Owl, which offer a direct language support, facilitate and encourage the use of the object-oriented paradigm.

Nowadays object-oriented programming is a subject that is still maturing. Techniques such as delegation and reflection have significant potential, but much work remains to be done for understanding and evaluating them completely. Furthermore, many current researchs aim to integrate more semantics to the object-oriented paradigm, bridging the gap between formal methods and object-oriented programming. According to Bar-David[11], the benefits of formal methods in object-oriented programming can be summarised as follows:

- unambiguous description of system bahaviour.
- uncriticisable criterion for judging the correctness of a particular implementation.
- there exist techniques for semiautomatically deriving correct implementation from formal specification.

There is a large and growing literature of formalism in programming, much of which can be applicable to object-oriented programming, such as, the use of type

	Ada	*C++*	*Eiffel*	*CLOS*	*Simula*	*Smalltalk-80*
Typing	static	static	static	dynamic	static	dynamic
Dynamic Binding	no	virtual	yes	implicit msg sending	virtual	implicit msg sending
Generic Classes	yes	yes	yes	no	no	no
Private Data	yes	yes	yes	yes	yes	yes
Private Methods	yes	yes	yes	yes	yes	no
Class Variables	yes	yes	no	yes	no	yes
Name of the Receiver	-	this	current	self	this	self
Access to Supermethod	-	::	renaming	call_next_method	this	super
Inheritance	no	single & mult.	single & mult.	single & mult.	single	single & mult.
Inh. Conflict Solving	-	explicit path	explicit renaming	linearisation	-	conflicts unauthorised
metaclass	-	no	no	standard class	no	metaclass class
other features	-	overloading generic functions	assertions	generic functions	active objects	any entity is an object

```
Key Words:
  'yes' means that the feature is present;
  'no'  means that the feature is not present;
  '-'   means that the feature is not applicable.
```

Table 2.4: Summary Table of the Features of Class-based Languages

specifications[97, 98], and algebraic specification[11]. Moreover, many investigations in the object-oriented community are also concerning the development of formalisms for object-oriented concurrent systems and metaobjects protocols.

As far as delegation is concerned, very few languages support both delegation and inheritance in a class-based framework. For instance, a good example of such a language is an object-oriented language called MUST[160] which is based on Smalltalk but includes extensions on the areas of delegation, multiple inheritance, and encapsulation. Moreover, some research has appeared concerning the issue of type-safety in delegation-based languages, such as the work by Agesen et al.[1], who have designed and implemented a type inference algorithm for the Self language.

2.3 Object-Oriented Design Methodologies

Usually the software process development includes the main following steps:

(i) stating a problem,

(ii) understanding its requirements,

(iii) planning a solution, and

(iv) implementing a program in a particular programming language.

A design methodology consists of building a model of a problem domain and then adding implementation details to it during the design of a system. The object-oriented approach allows the same concepts and notation to be used throughout the whole software life cycle, from *analysis* through *design* to *implementation.*

It is useful at this stage to define concepts such as method, technique and methodology. In the scope of this thesis, a *method* is defined as a set of systematic activities to carry out a task. A *technique* is the way to execute activities recommended by the methods, and *methodology* is a set of methods and techniques with which an objective can be reached[24]. There is some controversy on what characterises a software design methodology, and therefore what can be expected of one. Some researchers argue that many current design methodologies are merely notations for the expression of design; however, a good design methodology should comprise more than just a notation[24, 155]. Such methodology should provide guidelines

on the steps that should be followed at each stage during the development and should cover the whole software development life cycle.

While a design methodology can emphasise principles and provide guidelines for designers to follow, the act of designing a software is ultimately a creative process which requires skills, experience and common sense to be performed well. One important guideline that a design methodology provides is to hint at the way in which large and complex systems can be decomposed into smaller components which can be manipulated more easily. An object-oriented design methodology differs from a conventional one in the particular manner in which the system is decomposed into smaller parts and the nature of the relationships between them.

Although there is no standardisation across existing object-oriented analysis and design methodologies, there is a common distinction made between object-oriented analysis and object-oriented design. In general, analysis deals with the problem domain, and design with the solution domain. More precisely, *object-oriented analysis* models the problem domain by identifying and specifying a set of semantic objects that interact and behave according to system requirements. *Object-oriented design* models the solution domain which includes the semantic classes (with possible additions), and also interface, application and base/utility classes identified during the design process. Semantic classes define objects that have meaning to the problem domain; interface classes refer to objects that are associated with the user interface; application classes define objects that implement the control mechanisms for the system; and, finally, base/utility classes define objects that are application-independent. Object-oriented design is language independent, and precedes the physical design (implementation phase).

Many different object-oriented methodologies can be found in the literature. (An excellent discussion evaluating and comparing the current object-oriented analysis and design research can be found in [24, 112, 30].) There are a number of methodologies, methods and techniques, different notations and conflicting rules. Again, there is no standard representation of the object-oriented concepts amongst the different proposed notations. According to the evaluation presented in [112], two methodologies are particularly well ranked: Booch's methodology[20] and Rumbaugh et al.'s[131] Object Modelling Technique (OMT). Booch's work has been taken considerably further in a handbook on object-oriented design produced by an American software company. Meyer' book[105] also illustrates many important principles of object-oriented design with examples written in the Eiffel programming language. In particular the notion of "design by contract" is very important to develop large, reliable programs, providing practical guidelines enforced by the use of preconditions, postconditions and automatic inheritance of assertions in Eiffel. For the purposes of this research, we have chosen the OMT

methodology mainly because we felt that it encompasses (as many other methodologies) the main object-oriented concepts coherently and its notation was simple to understand. Below we describe this methodology in more detail.

2.3.1 Overview of The OMT Methodology

Object Modelling Technique (OMT)[131] is a language-independent graphical notation that represents a set of object-oriented concepts. The OMT methodology has three basic stages, which are referred to as *analysis*, *system design* and *object design*. Each stage is now discussed in turn.

The *Analysis Stage* concerns with understanding and modelling the application software and the domain within which it iterates. The goal of this stage is to understand the application in terms of classes, using the object, dynamic and functional models to represent the properties meaningful to the application. Each model is now defined:

The Object Model describes the aspects of a software system in terms of classes, in an object model diagram. The object model is represented by graphs whose nodes are classes and arcs denote relationships of specialisation, aggregation, or any association between classes.

The Dynamic Model describes the aspects of a software system which may change over time due to events, and is used to understand the control flow of a software system. The dynamic model is represented by familiar state transition diagrams whose nodes are states and arcs are transitions (caused by events) between states.

The Functional Model describes the data transformations within a software system and employs the well-known data flow diagrams to represent computations of output values from input values.

The object, dynamic and functional models are orthogonal parts of the description of the whole system. However, the object model is the most fundamental, because it is necessary to describe *what* is changing before describing *when* and *how* it changes. The output from analysis is the *Analysis Document* that consists of the problem statement, object model, dynamic model and functional model. From this document it is possible to identify: objects and their attributes, operations, the visibility of each object in relation to the others, and the interface of each object.

The *System Design Stage* focuses on decisions about the high level structure of a software system. In this stage, the software system is divided into subsystems, without in fact employing any concepts related to the object-oriented paradigm. Using the object model as a guide, the following steps are performed:

1. organize the system into subsystems.
2. identify the concurrency inherent in the problem.
3. allocate the subsystems to processors and tasks.
4. choose the basic strategy for implementing data stores.
5. identify the global resources and determine the mechanisms for controlling access to them.
6. consider the boundary conditions.
7. establish the trade-off priorities.

The output from system design stage is the *System Design Document* that defines the structure of the basic architecture for the system as well as the high-level strategy decisions.

Finally, *Object Design Stage* is targeted at the data structures and algorithms need to implement each class in the object model. During this stage, dynamic and functional aspects are combined and refined, and more details about the control flow of a software system are defined. The following steps are performed:

1. obtain operations for the object model from the other models.
2. design algorithms to implement the operations.
3. optimize access paths to data.
4. adjust class structure to increase inheritance.
5. design implementation of associations.
6. determine the exact representation of object attributes.
7. package classes and associations into modules.

The document resulting from the object design stage is called the *Design Document*, and contains the detailed object model, the detailed dynamic model, and the detailed functional model.

2.3.1.1 Object Model

The first step in analysing the application requirements is to construct an object model. The object model shows the static data structure of the real-world system. The object model precedes the dynamic model and the functional model because the static structure is usually better defined and easier for humans to understand. The following steps are performed in building an object model:

1. identify object classes.
2. prepare a data dictionary containing descriptions of classes, attributes, and associations.
3. add associations between classes.
4. add attributes for objects and links.
5. organize and simplify object classes using inheritance.
6. test access paths using scenarios and iterate/refine the above steps as necessary.
7. group classes into modules, based on close coupling and related function.

The object model can be summarized in the following expression:

$$\boxed{\text{Object Model} = \text{Object Model Diagram} + \text{Data Dictionary}}$$

2.3.1.2 Dynamic Model

The dynamic model shows the time-dependent behaviour of the system. It is important for interactive systems. The aspects of a system that are concerned with changes to the objects and their relationship over time are captured in the dynamic model, in contrast with the static model (object model).

The major dynamic modelling concepts are *events*, which represent external stimuli, and *states*, which represent values of objects. The *state transition diagram* is a graphical representation of state machines. The use of events and states are emphasised to specify control. Moreover, states and events can be organised into generalization hierarchies to share structure and behaviour. It uses the notation

of David Harel[65] for drawing *structured state diagrams* using nested diagrams to show structure.

In summary, the following steps are performed in constructing a dynamic model:

1. prepare scenarios of typical interaction sequences.
2. identify events between objects and prepare an event trace for each scenario.
3. prepare an event flow diagram for the system (message change).
4. build a state diagram for each class that has an important dynamic behaviour.
5. check for consistency and completeness of events shared among the state diagrams.

The dynamic model can be summarized as follows:

Dynamic Model = State Diagrams + Global Event Flow Diagram

2.3.1.3 Functional Model

The functional model shows how values are computed, without regard for sequencing decisions, or object structure. It shows which values depend on which other values and the functions that relate them. The following steps are performed in constructing a functional model:

1. identify input and output values.
2. build data flow diagrams showing functional dependencies.
3. describe what each function does.
4. identify constraints.
5. specify optimisation criteria.

The functional model can be summarized as follows:

Functional Model = Data Flow Diagrams + Constraints

2.3.2 Reusability and Extensibility

Some of the advantages of object-oriented programming languages are their support for encapsulation, reusability and extensibility. *Encapsulation* is the strict enforcement of information hiding (see Section 2.2.1.5). *Reusability* is the ability of a system to be reused, in whole or in parts, for the construction of new systems. *Extensibility* is the facility with which a software system may be changed to account for modifications of its requirements[109]. These three aspects are particularly relevant to the development of large software systems.

Reusability is the practice of incorporating existing software components into software systems for which they were not originally intended. It is likely to be more cost effective spending some time searching for a reusable component rather than defining, implementing and testing a new component. In the past, the idea of reusability was associated with source code reuse or invoking subroutines from a library[53]. Thus, software reuse was usually carried out at the implementation phase. However, reuse of source code during the implementation phase is a very limited kind of reusability. Greater benefits are obtained when reusability is performed at higher levels.

Some investigations have been considering how to reuse parts of a software specification and design[34, 76]. Design reuse is fundamental to successful reusability under the object model. At a higher level than implementation, reusability involves a classification of software components which gives the information on what each component does, and accessibility which allows a component to be searched for, retrieved and hence reused.

Experiences so far have proved that design for reuse is not straightforward in practice. Most of the obstacles have concentrated on the problems of classifying the components, searching for potentially reusable components and accessing libraries of reusable components. Johnson and Foote[79] emphasise that the object-oriented paradigm is not a panacea for reusability. They argue that software reusability does not happen by chance and designers should plan to reuse, and new classes should be designed for reusability. Tracz[152] provides some insight on the interaction between reuse and object-oriented design, and states that:

(i) Software reuse is both a technical and non-technical problem involving psychological and economic barriers.

(ii) Domain analysis can play a role in solving the reuse problem.

(iii) Designing software from reusable parts is not like designing hardware using available integrated circuits.

(iv) Reusing software that was not planned for reuse is harder than reusing software that was designed for reuse.

(v) Software reuse will not just happen.

Nowadays the significant cases of reuse involve frameworks rather single components[23]. Objects and classes are building blocks that are just too small to achieve high levels of reusability. On the other hand, frameworks are bigger building blocks which define collections of collaborating classes that capture both the small-scale patterns and major mechanisms that, in turn, implement the common requirements and the design in a specific application domain. A framework is thus viewed as a codified architecture for a problem domain that can be adapted to solve specific problems. A framework makes it possible to reuse an architecture together with a partial concrete implementation. A class can use abstract operations to describe the framework of a reusable abstraction, and the functions can be filled in by the class user in the context where the abstraction is to be applied.

As asserted by Firesmith[51], today's object-oriented development methods and languages will remain insufficient until they support frameworks. The major current object-oriented development methods focus on developing application (and classes and objects) largely from scratch. In the context of object-oriented programming, apart from Eiffel which supports the notion of *clusters*, no major object-oriented languages support a building block larger than a class. Besides, it is also necessary the ability to document these higher-level abstractions and responsibilities so that the semantics of these larger building blocks can be captured. Some investigations have been reported on the literature related to this topic (see [22, 81]).

Another approach to software reuse is the concept of a *design pattern*[56, 57, 13]. Design patterns capture the intent behind a design by identifying objects, their collaborations, and the distribution of responsibilities. They form a base of experience for building reusable software, and act as building blocks from which more complex designs can be built. Gamma *et al.*[56] reports on how to express and organise design patterns, introducing a catalog of them. An important distinction between frameworks and design patterns is that frameworks are implemented using a programming language while patterns are ways of using a programming language. In this sense, frameworks are more concrete than design patterns.

Design patterns are also related to *idioms* introduced by Coplien[34]. These idioms are concrete design solutions (in fact in the context of C++). In contrast to

idioms, design patterns are more higher-level abstractions, trying to abstract design rather than programming techniques. Design patterns are largely inspired on the influential work on "patterns" by Christopher Alexander[5]. Lea[90] discusses the importance of patterns in object-oriented design. He states that patterns can raise the expressiveness and level of description supported by object-oriented constructs, and conversely, object-oriented techniques can strengthen pattern-based design notions through concepts such as inheritance, delegation and reflection.

Discussions of the benefits of object-oriented design often emphasise reuse rather extensible, maintainable systems. Component reuse plays an important role in the maintenance of the system, which is by far the most expensive part of the life cycle of a system. Maintenance is a process of reusing components across time, rather than across applications[12]. Enhancing the system results in making modifications to the system components in unforeseen ways, just as during system reused components need to be altered to fit the new applications.

Some authors, such as Haythorn[73], argue that all discussion about reuse is a marketing mistake. Once we are really interested in building systems, maintainability should be a more fundamental aspect than reuse. In fact, the maintainability aim does not conflict with the reuse aim, and the first step towards reuse is to design for maintainability. A maintainable object-oriented design is constructed around a central set of classes which are characterised more by the variation they hide rather than what they are. In this sense, many of the current design methodologies cannot be expected to produce truly maintainable object-oriented design. Essentially to building systems that are robust under changes, it is necessary to anticipate and consider expected changes. Whatever method or notation you use, two additional aspects should be added to analysis and design: (i) the listing of expected changes, and (ii) the critical examination of the initial design in terms of its robustness to these changes.

The role of maintenance in design is described by Winograd and Flores[159]:

> "... to anticipate the forms of breakdown and provide a space of possibilities for action when they occur. It is impossible to completely avoid breakdowns by means of design. What can be designed are aids for those who live in a particular domain of breakdowns."

We will return to this point later in Chapters 3 and 4, focusing on object-oriented design for fault tolerance.

2.4 Exceptions in Object-Oriented Languages

Exception handling is an essential feature if strong typing is to be meaningfully integrated into a useful object-oriented language. If we accept the view that any object is essentially an entity that provides a service to client objects, and that an object type is a description of the "contract" between client and server objects with respect to these services, then without a mechanism of exception, as an integral part of that contract, there is no way for an object to notify its clients when the contract cannot be honored.

Experience with the integration of an exception mechanism with the object model is still restricted. Such an integration should be coherent with the structuring principles of the object model. The basic executing unit in the object model is an object's operation invocation. If we accept the view that any object is essentially a component that provides services to client objects, an exception becomes an abnormal response of an operation invocation when the service cannot fulfill its requirements. This solution fits naturally with the object model.

We therefore now revisit some important design issues of the classical exception mechanism[61, 136] in the context of object-oriented languages, such as:

- How to represent exceptions in an object-oriented system.
- How to declare an exception and what is its scope.
- How to relate exception raising to interface checking.
- How to declare default handlers.
- What is the impact of inheritance on all these points.

In the remainder of this section, first we propose a taxonomy which is used to evaluate and compare designs of exception handling mechanisms implemented in several prominent object-oriented programming languages. Second, we survey exception handling mechanisms in several object-oriented systems. Finally, this is followed by a discussion of exceptions in object-oriented systems.

2.4.1 A Taxonomy for Exception Handling Systems

For the purposes of this review, we classify the several approaches for implementing exception mechanisms found in the literature in two dimensions: exception

representation and placement of handlers. Exception representation refers to how an exception is specified, and placement of handlers indicates the different positions that a programmer can choose to bind an exception to a handler. The first dimension is subdivided into the following categories:

Exceptions as Full Objects. Exceptions are hierarchically organised classes. To raise an exception is to create an instance of the related class, then to call it with a raise method.

Exceptions as Data Objects. Exceptions are classes and an instance of one of these classes is created every time an exception is raised. To raise an exception is to pass an exception object as a parameter to a handler. The signaller of an exception is allowed to pass any kind of parameter by declaring them in the state of the exception object.

Exception as Strings. This is the classical approach adopted by most imperative languages, such as CLU[95] and Ada. Exceptions are implemented as string variables. Raising an exception sets this string variable and returns the control to the caller, which is in charge of testing the variable.

The second dimension, which is related to the placement of handlers, is arranged in the following groups:

Exception Handlers. Such handlers are associated with exceptions themselves and are always invoked when no more specific handlers can be found. They are the most general handlers and must be valid in any case. They are independent from any execution context as well as from object state. For instance, an exception handler could print an error message or make a general correction action.

Class Handlers. These are attached to classes. In this way, they allow one to define a common behaviour for a class in exceptional situations. Examples where class handlers are useful could be applications such as debugging or context restoration.

Statement Handlers. These are attached to blocks of instructions allowing context-dependent responses to an exception.

Object Handlers. These are bound to variables in declarations; that is, each data object declared has its own set of (exception,handler) binding pairs specified in its declaration.

According to the taxonomy above discussed, a classification of the reviewed approaches is shown in Table 2.5. Each design is now discussed in some detail.

	Exceptions as Full Objects	*Exceptions as Data Objects*	*Exceptions as Strings*			
	Dony	C++	Modula-3	Guide	Eiffel	Cui
Exception Handlers	yes	no	no	no	no	no
Class Handlers	yes	no	no	yes	yes	yes
Statement Handlers	yes	yes	yes	yes	yes	yes
Object Handlers	no	no	no	no	no	yes

Key Words:
'yes' means that the feature is present;
'no' means that the feature is not present.

Table 2.5: Summary of Exception Handling System Features

2.4.2 Survey of Object-Oriented Exception Handling Systems

Now we present the most well-known object-oriented exception systems in order to evaluate their main strengths and weaknesses. The coverage of these approaches is deliberately concise and the references can be used to provide additional information.

2.4.2.1 Dony's Approach

A complete object-oriented representation of exceptions has been developed by Dony[43, 44], and it has been implemented in Smalltalk-80 and also in an object-oriented programming environment called SPOKE[14]. An important characteristic of this approach is the object-oriented representation of exceptions. Exceptions are hierarchically organised classes. To raise an exception is to create an instance of the related class, then to call it with a raise method.

In most imperative languages, exceptions are usually represented as strings of the type "exception" that cannot be inspected. However, if exceptions are represented as classes, some improvements can be achieved, such as:

- exception occurrences are class objects that can be inspected, modified or enhanced,
- new exceptions can be easily defined,
- exceptions can be organised in a hierarchy based on common behaviour, which makes the system extensible and reusable,
- all predefined properties are reusable via inheritance and method overriding,
- handler definition is powerful, since handlers treat not just one kind of exception but all exceptions that are subclasses of it, and
- handlers that are independent of any execution context can be defined as methods on exception classes.

As stated before, in this approach exceptions are classes; an instance of an exception X is created each time X is raised. The definition of a new exception corresponds to the definition of a new class. Handlers are methods or instances of a specific class named *protected-handler*. They can be attached to statements, to classes and to exception classes. Handlers attached to classes are called *default* handlers.

Signal is a method defined in a root class called *Exception*, and it is a redefinition of the method *new* that first creates an instance of an exception and then searches a handler for it. All handlers have a unique parameter automatically bound at handling time to the instance of the current exception and through which arguments provided by the signaller are conveyed.

Handling is only performed via message sending to the exception object and all protocols for handling an exception are defined as methods in the *Exception* class and inherited. Four basic ways of handling an exception are provided: resumption, termination, signalling a new exception or propagating a trapped one. When handlers do not explicitly explicitly one of these options, the exception *ExceptionNotHandled* is signalled. Termination means that the method activation that raises the exception is exited. Resumption means that after the execution of the handler, control returns to the statement following the signalling one.

When an exception is raised, the handler search proceeds as follows:

(i) First handlers that are attached to commands that dynamically include the signalling one are searched. The search stops as soon as a handler, whose parameter type is a supertype of the signalled exception, is found.

(ii) If none is found, the system tries to find default handlers attached to the class or upper classes of the signalling active object;

(iii) If none is found, the system looks for default handlers attached to the signalled exception itself.

(iv) If none is found, the exception is then propagated to the operation caller, and the sequence of steps is again repeated.

This approach is interesting because it tries to integrate exceptions into the standard invocation mechanism. Nevertheless, by doing so it complicates its semantics. The behaviour of the termination, retry and resumption policies are all defined as methods in a root *Exception* class. As a consequence, a *raise* method call can eventually resolve into a *return* (termination), a normal call (resumption) or something more complicated (retry).

2.4.2.2 Exception Handling in Eiffel

The exception mechanism of Eiffel is integrated with the notion of *design by contract* earlier presented in 2.2.3.2. The contract of a software component defines the observable aspects of its behaviour, i.e., those which its clients expect. An exception is the occurrence of an event which prevents a component from fulfilling the current execution of its contract.

In Eiffel, contracts are expressed through assertions: preconditions, postconditions and class invariants. If an assertion attached to a method is found to be violated, an exception is raised in the routine currently being executed. When an exception occurs during the execution of a routine, its execution is stopped and a handler, termed a *rescue* clause, will be executed instead. Rescue clauses must restore the class invariant by either retrying the operation if the precondition still holds (resumption) or reporting failure to the caller by reraising the exception (organised panic).

A routine with no rescue clauses will be considered to have an empty rescue clause, so that all exceptions will cause immediate failure of the routine. However,

a rescue clause may be included at the class level, and will then be used by any routine of the class which does not have its own clause.

The organised panic policy should restore the class invariant, addressing the problem of object consistency after the occurrence of an exception. The formal version of this requirement is that any branch of a rescue clause not terminating with a retry should produce a state satisfying the invariant, independently of the state in which it is triggered.

The Eiffel model is based on the contracting metaphor: under no circumstances should a routine pretend it has succeeded when in fact it has failed to achieve its purpose. A routine may only succeed or fail, there is no intermediate ground. Eiffel's exception mechanism is thus more restrictive than the exception mechanism built into other existing languages. In contrast, the traditional exception mechanism, such as in Ada or CLU, is based on the *raise* instruction, which signals an exception explicitly, cancels the routine that executed it and returns control to its caller. The caller may handle the exception or, if it has no such handler, will itself return control to its caller, but there is no rule as to what a handler may do.

2.4.2.3 Exception Handling in Modula-3

Modula-3's exception handling mechanism[27] is based on a semantic model similar to CLU and Ada. Exceptions are implemented as string variables and are propagated upward through nested dynamic contexts, looking for a handler. Handlers are attached to a block of instructions by means of what is termed a *try* statement.

If, during the execution of a *try* body, one of the listed exceptions is raised, execution ceases, and control passes to the corresponding handler body. If the exception raised does not match any listed, and an *else* clause is present, the *else* handler body receives control. If no *else* clause is present, a handler is sought in the statically enclosing context (a *try* statement may be nested in a *try* body). If no handler is found there, the search continues in the context of the calling procedure. If no handler can be found, the computation is suspended and the debugger receives control.

Before a handler is executed, all dynamic contexts between the raiser and the *try* body (inclusive) are finalised: the stack is unwound, register values are restored, and explicit finalisation actions are invoked (if what is termed a *finally* clause is specified). When the handler has finished its execution, control passes to the

statement following the *try* construct, exactly where it would have gone if the *try* body had not encountered an exception.

2.4.2.4 Exception Handling in C++

The model proposed in C++[84] introduces an exception mechanism that is sensitive to context. The context for raising an exception is termed a *try* block. Handlers are declared at the end of a *try* block using the keyword *catch*. An exception can be directly raised in a *try* block by using the *throw* expression. The exception is handled by invoking an appropriate handler selected from a list of handlers found immediately after the handler's *try* block.

In general, exceptions are values, but they can be declared as classes and an instance of one of these classes is created every time an exception is raised. To raise an exception is to pass an exception object as a parameter to a handler. The handler declares its parameter as being of a given class, but may catch exception objects of any subclass. The signaller can pass any kind of parameter by declaring them in the state of the exception object.

So it is possible to apply the mechanisms of inheritance and subtyping to build a hierarchy of related exceptions and then opt to handle these exceptions individually or as a group. Syntactically an exception specification may be part of a function declaration (in C++ it is optional), giving information about the types of exceptions that the function can throw.

If the exception mechanism fails at runtime for some reason, for example, no handler is found for an exception, the system-provided handler *terminate()* is called. By default the *abort()* function is called, but the programmer can use the *set_terminate function* function to provide a handler.

2.4.2.5 Exception Handling in Guide

All the languages discussed so far have kept the association of a handler with a block of instructions, syntactically as well as semantically. A handler (with a termination policy) resumes the execution at the instruction following the block. In Guide[87], a handler may be semantically associated only to a method call, and not to a block of instructions. This model allows a clear separation of the exception handling code from the main algorithm.

Thus, a handler is semantically associated to method invocations. The normal

continuation after the execution of a handler is from the point just after the raising method invocation. It has nothing to do with the syntactic declaration of the handler. In order to help the user precisely define the scope of his/her handlers the system allow a handler to be associated not only to an exception name but also to a type and method name, that is, handler declarations can be syntactically factorized at the method or class level.

The nature of Guide exceptions are strings. An exception is attached to a method, and more generally to the type where it is declared. Exceptions potentially raised by a method appear in its interface, and they must be included in the interface (it is not optional like in C++).

Guide's model is based on the termination policy. So the calling object does not answer to the raising of an exception. Raising an exception exits the method in the same way as a return statement does. The *retry* policy is provided through the *retry* keyword that only a handler can use. The Guide system provides a default handler which propagates the *uncaught_exception* system ensuring that an exception will either be handled or will eventually terminate the task.

2.4.2.6 Cui's Approach

Cui's approach, called data-oriented exception handling[37], is a design that associates handlers with data objects in their declarations. Exceptions are associated with type declarations in generic package specifications, and handlers are bound to variables in declarations. Each data object declared has its own set of (exception, handler) binding pairs specified in its declaration. Three language features are defined to implement this design: *#exception*, *#when*, and *#raise*. Exceptions are declared by attaching an *#exception* clause to the type exported from the specification part of a package. Handlers are associated with data object's declaration by attaching a *#when* clause to the declaration that specifies handler procedures for the exceptions defined on the data object's type. Exceptions are signalled by *#raise* statements that transmit parameters, indicating the object with failure. Default handlers for exceptions can be specified in a type declaration and inherited by variables declared with that type.

This concept has been implemented with an Ada preprocessor and empirical studies have shown that its use can produce programs that are smaller and better structured when compared to the programs produced using Ada's traditional exception handling. In Ada's exception handling mechanism, although handlers appear after the main algorithm, introducing blocks in the middle of a statement list to associate different handlers with different objects inserts exception handling

code in the middle of the main algorithm preventing a clear separation between them. The data-oriented exception handling removes exception handling code from algorithmic code helping code writability and understanding.

2.4.3 Discussion

Language features for exception handling continue to evolve. Most of the existing object-oriented languages have adopted the classical exception handling design. Their main contributions consist of a strict control of the exceptions a method may raise or propagate, and an innovative representation of exceptions as objects. Moreover, the ability of factorising handlers at method and class levels promotes a better code writability and program structuring.

Undoubtedly, the use of inheritance and polymorphism to implement an exception mechanism has a number of advantages when compared to the classical approach, such as:

- the choice of designing exceptions as classes enables them to be organised into a hierarchy which makes the system extendible and reusable,
- handler definition is powerful, since handlers do not only handle one kind of exception but all exceptions that are subclasses of it,
- handlers that are independent of any execution context can be attached to exception classes, and handlers attached to classes can be inherited by subclasses.

Dony's approach is very interesting, although it seems not to have a simple semantical model since the behaviour of the termination, retry and resumption policies are all defined as methods in a root *Exception* class. As a consequence, a *raise* method call can eventually resolve into a *return* (termination), a normal call (resumption) or retry. Meyer, for instance, advocates that a good exception handling mechanism should be simple and modest. In this sense, Dony's approach is complicated, and, in my opinion, it needs to be restricted to be a useful tool for programming.

Exceptions in Eiffel are handled in a very peculiar way. The only authorised policies are either retry the execution of the whole body or to propagate the exception. Although the tool is not sophisticated it fulfills its goal of providing consistency

of objects. The rescue clauses ensure the correctness of the postconditions of the method, which includes the invariant of the object.

The mechanism offered in Modula-3 is similar to the classical approach implemented by languages such as Ada and CLU. Its main contribution is a strict control of the exceptions a method may raise or propagate; that is, operations specify exceptions as part of their signatures. C++'s approach has, compared to this, a major improvement of representing exception as objects. The model is simple and flexible, however nothing is done to ensure consistency of objects as in Guide.

In Guide, the *restore* keyword allows one to define a restoration block which is executed whenever the method exits abnormally (raises an exception). The restoration block is not executed if the method returns normally. Moreover, Guide's proposal helps the user to define more precisely the scope a handler (a handler is semantically associated to method invocations and a handler can be associated to a type). The main limitation of Guide's approach is that it represents exceptions as simple strings associated to a type. A solution discussed in [87] is to organise the exceptions in a hierarchy respecting the subtype hierarchy of the corresponding types.

The data-oriented exception handling proposed by Cui, in which a handler can be attached to data objects, naturally fits in an object-oriented approach and it should be explored (if a handler can be attached at method, class and exception class level, why not can it be attached at object level?) Although this model is implemented in an object-based language (Ada), it can be implemented in an object-oriented language.

As mentioned earlier, the integration of an exception handling with the object-oriented paradigm is still evolving. In particular, concurrent object-oriented systems is an important area within object-oriented programming, and many intriguing possibilities exist in moving objects into a concurrent world - allowing several objects to execute concurrently and/or allowing each object to execute several of its methods concurrently. Some researchers have been investigating means to integrate exception handling with parallel object-oriented programming, such as Issarny[78] with the implementation of the strongly-typed parallel object-oriented language called Arche. This language supports a mechanism based on a parallel exception-handling model whose features enforce the construction of correct and robust programs.

2.5 Object-Oriented Fault Tolerance: A Preview

The primary goal of this thesis is to provide fault tolerance to software systems - in particular with regard to environmental faults - exploiting object-oriented concepts. Object-oriented programming techniques provide the required framework for the construction of useful abstractions for building up fault-tolerant program structures. They incorporate subclassing, delegation, and compositional constructs, and further out, metalevel reasoning constructs, that although by no means so completely established and accepted, allow the creation of useful systems based on transparency, reconfigurability and flexibility. As far as fault tolerance is concerned, exception mechanisms within an object-oriented framework are very important, not least since software is prone to design faults. In the next chapter, we discuss an approach for the provision of object-oriented environmental fault tolerance. The scope of discussion also encompasses a brief discussion of object-oriented hardware fault tolerance and software fault tolerance for completeness.

2.6 Conclusions

This chapter has investigated the main trends in the object-oriented arena and introduced a dictionary of the object-oriented terminology employed in this thesis. This has been necessary because there have been no generally accepted terminology and definitions of what these various terms mean. The object-oriented field is so divided in terminology, as well as in definitions of concepts, that many surveys of relevant object-oriented concepts are really confusing, obscure and inadequate. As pointed out by Nelson[117], one might even say that we have created something of an object-oriented "Tower of Babel". In fact, in my opinion, most of the existing analyses and surveys found in the literature had caused much more confusion than clarification. Moreover, the concepts of object-oriented programming were presented in language-independent manner, encompassing the essence of the object-oriented programming and making it simple for understanding what it is all about.

By no means, the explanations given here are meant to be definitive - as we had already mentioned, there are still substantial disagreements within the object-oriented community with respect to terminology and also definitions of concepts. In spite of that, we hope to have chosen a clear and consistent set of definitions for use in our research. It has been hard work to examine the object-oriented

literature and try to make a compromise between the many different trends and suggestions, especially concerning the use of conflicting notations and definitions. In spite of that, we hope to have contributed towards the clarification of the most important object-oriented concepts.

BLANK PAGE
IN
ORIGINAL

Chapter 3

Object-Oriented Fault Tolerance

> "The larger the system, the greater the probability of unexpected failure."
>
> "Complex systems usually can fail in an infinite number of ways."
>
> "Complex systems have complex behaviors."[54]

Various sources of complexity can be identified in a system, but this thesis concentrates on faults (especially environmental faults), and the frequent need to cope with them as an unvoidable source of complexity, but one which nevertheless has to be minimised so as to keep under control the system complexity. In other words, this chapter discussed the impact of faults on system complexity and structuring, emphasing the treatment of faults/major changes in the behaviour of the environment entities. However, a more thorough and encompassing approach would not only consider the treatment of environmental faults but also the treatment of design and hardware faults so as to cope with all different kinds of faults in the overall system.

In what follows we will address each of these faults separately in an object-oriented framework. First, we discuss briefly some investigations which have considered object-oriented techniques for the construction of hardware fault tolerant systems. Second, we present an object-oriented approach for the provision of software fault tolerance. As far as software fault tolerance is concerned, very few research has been reported on the application of object-oriented techniques for the provision of software fault tolerance. Finally, we show an object-oriented approach for the provision of environmental fault tolerance[129]. As stated before, the main topic of the thesis is to deal with environmental faults and their impact on system

complexity, and the mention of design and hardware faults is just for addressing the issue of completeness.

3.1 Fault Tolerance Concepts

Many applications often demand high reliability and availability requirements from computer systems, and, therefore, the inclusion of fault tolerant techniques in such applications is justifiable since faults are likely to occur and users expect that they can be tolerated without the detriment of system functionality and reliability. In general, fault tolerance is based on the provision of redundancy, both for error detection and error recovery. Fault tolerant techniques attempt to prevent faults from causing system failures. Usually the steps performed to design and implement a fault tolerant systems are: error detection, damage assessment, error recovery, and fault treatment[91].

Hardware redundancy is routinely used to enhance reliability/availability. However, software is a major component in computer systems, and thus tends to become the reliability bottleneck. The incorporation of software fault tolerance in systems requires a structured and disciplined approach. Design diversity as a means of achieving fault tolerance in software has been suggested by several authors.

Following the terminology used by Lee and Anderson[91], a *system* consists of a set of *components* which interact under the control of a design which is itself a component of the system[91]. The system model is recursive in the sense that each component can itself be considered as a system in its own right. Components receive requests for service and produce responses. If a component cannot satisfy a request for service, it will return an exception. At each level of the system, a component (called an idealised fault-tolerant component) will either deal with exceptional responses raised by components at a lower level or else propagate the exception at a higher level of the system.

Type of Faults

At this point, we remind the reader that in the scope of this research we consider three categories of faults: *hardware*, *design* and *environmental* (see Section 1.1). Many studies have been reported by others on the construction of hardware fault-tolerant systems based on object-oriented techniques, and we briefly mention some of them in Section 3.2. However, few researches have explored the construction of software fault-tolerant systems based on object-oriented concepts,

and in Section 3.3 we discuss some ideas developed by our group here in Newcastle. In Section 3.4, we discuss the main contribution of this thesis, that is, an object-oriented approach for providing environmental fault tolerance. In the reminder of this Section, we briefly review some important concepts related to fault tolerance, such as, exception handling, idealised fault-tolerant component, recovery block, etc.

3.1.1 Exception Handling and Fault Tolerance

Exceptions and exception handling mechanisms are needed, in general, as means of connecting actions of system components that belong to different levels of abstraction. As mentioned above, if a component cannot satisfy a request, an exception is returned. So the responses from a component can be classified into two categories: *normal* and *abnormal*. In software systems, the abnormal responses are usually referred to as exceptional responses or simply exceptions. The activity of a component can be divided into two parts: a *normal* part and an *exception handling* part. The normal part implements the component's normal service while the exception handling part implements the measures for tolerating faults that cause such exceptions.

Signalling an exception results in the interruption of the normal processing, followed by the search and invocation of a handler to deal with the exception. Exceptions can be categorised into two groups: *interface exceptions* which are signalled in response to a request which did not conform to the component's interface, and *failure exceptions* which are signalled if a component determines that for some reason it cannot provide its specified service.

When the exception handler terminates normally, the exception is said to be handled. The system then can return to normal operation; however, there is an issue whether the internal activity of the component can be resumed after the exception has been handled by the system. In the *termination model*, execution will continue from the point at which the exception was handled, not the point at which it was raised[95], i.e., the execution continues from the first statement followihg the point of call that signalled the exception. In the *resumption model*, the handler has the capability to resume the internal activity of the component after the point at which the signal was raised. Most of the work in exception handling is for imperative languages such as, Ada, CLU, and Mesa. However, as discussed in Chapter 2, exception handling is also a feature of object-oriented languages. For example, exception handling in C++ is based on a termination model. A formal treatment of exceptions and exception handling is given by

Cristian[36].

3.1.2 Idealised Fault-Tolerant Component

An *idealised fault-tolerant component* is a well-defined component whose interface exceptions are signalled in the normal part of the component, while failure and interface exceptions from sub-components invoke the exception handling part of the component. If these exceptions are handled successfully, the component can return to providing normal service. However, if the component does not succeed in dealing with such exceptions, it should signal a failure exception to a higher level of the system.

In software fault tolerance, exception handling is mainly used to implement idealised fault-tolerant components which are software objects able to return well-defined and foreseen answers, whatever may happen while they are active, even when an exceptional situation occurs.

3.1.3 Fault-Tolerant Software Techniques

Fault-tolerant software techniques include N-version programming[10] and recovery blocks[126]. The former uses voting on the results of various versions for error detection, and the latter uses an acceptance test and rollback recovery. N-version programming uses extra software components of diverse design called *variants* so that software errors are masked from the environment of that system.

A recovery block consists of a primary module, one or more alternates, and an acceptance test. The primary and the alternate modules are based on different algorithms for the same problem and may be implemented by different programmers. On a given input data set, the primary is executed first and the results are checked using the acceptance test. If the acceptance test does not accept the results, a rollback recovery is attempted and this process is repeated for each alternate module in succession until either the rollback recovery fails or a module is found to produce results that are accepted by the acceptance test or until all modules have failed to satisfy the acceptance test. In the latter case, the recovery block is said to have failed on this input data set.

The N-version programming scheme can be considered as an immediate extension of NMR structures used in hardware[91]. N versions of a program which have been independently developed to satisfy a common specification are executed and

their results compared by some form of replication check. Based on a majority vote, this check can eliminate erroneous results and pass on the (presumed to be correct) results generated by the majority to the rest of the system.

3.1.4 Error Recovery

A fault in a system component may cause an error in the internal state of the system which eventually can lead to the failure of the system. Two approaches are available for eliminating the errors from the system state, known as *forward* and *backward* error recovery. Forward error recovery attempts to correct an erroneous state by making selective corrections to remove errors, while backward error recovery restore a previous state of the system, which is (at least temporarily) presumed to be free from errors.

Exception and exception handling constitute a common mechanism applied to providing forward error recovery. In contrast, the recovery block scheme provides a system structure which supports backward error recovery. Thus, exception handling and recovery block are usually known as complementary approaches for achieving error recovery of a system.

3.2 Object-Oriented Hardware Fault Tolerance

The use of the object-oriented model to system design can reduce system complexity by allowing a software to be decomposed into a set of cleanly separated components. Furthermore, if these components are chosen carefully, they may be used in the construction of several related applications, that is, they can be reused. In the literature many researches have demonstrated that it is possible to build reusable components using object-oriented techniques that address hardware fault tolerance, such as network and computer failure, especially concerning distributed applications, and here we do not aim to provide a survey of this topic.

The realisation of hardware fault tolerance can be done in many different ways. One approach, used by systems like Avalon/C++[41] and Arjuna[138], consists of using properties of object-oriented languages, such as inheritance, to make objects recoverable. In particular, the Arjuna distributed programming system[138] permits the creation of fault-tolerant, distributed applications, and consists of a number of tools, such as a reliable RPC mechanism, an RPC stub generator, and C++ classes providing recovery, concurrency control, persistency and atomic

actions. For example in Arjuna objects can inherit persistency or atomicity characteristics, and no changes were made to the C++ compiler or runtime system.

However, these researches have also highlighted a number of inadequacies in the support provided by many conventional object-oriented programming languages for building components that address hardware fault tolerance. In particular, this often requires an extension to the language (by means of a pre-processor or a changed compiler) and/or the adherence by the programmer to a set of programming conventions that obscures the functionality of the application. Ideally the provision of such reusable components should be transparent rather than intrusive to the programmer.

So an alternative solution for the implementation of hardware fault tolerance in a reusable fashion is to use reflection transparently at the metalevel. Usually dependability mechanisms and measures are more concerned with the virtual machine that executes the application rather than the application itself. In this sense, reflection can be applied to extend the semantics of the programming system in a transparent and clear form to the application programmer, promoting a clean separation of concerns between the functional and non-functional requirements.

It seems that reflection is a technique with potential; however, more work is required to investigate how these ideas can be arranged in a coherent framework applied to the provision of dependability[148]. Some studies have already appeared, such as the work by Fabre *et al.*[49], which shows how reflection and object-oriented programming can be used to ease the implementation of classical fault tolerance mechanisms in distributed applications. The authors have claimed that the use of reflection can help to improve the transparency of fault tolerant mechanisms to the programmer, and presented the implementation of some classical replication techniques using the reflective object-oriented language Open-C++[31].

3.3 Object-Oriented Software Fault Tolerance

The preceding section has discussed some object-oriented approaches to the tolerance of hardware faults, and the subsequent one will deal with the construction of fault-tolerant components for dealing with environmental faults, using a scheme of parallel hierarchies of components linked together via a delegation mechanism. This section discusses an object-oriented approach for the provision of software fault tolerance. As we pointed out before, this topic has been addressed by very

few studies. This was the main motivation for developing the work described in this Section, which was carried out in collaboration with the colleagues Robert Stroud, Jie Xu and Brian Randell. A more detailed presentation of this discussion, on which this material is largely based, can be found in [130], [127], [128] and [163].

The objective of this research is the exploitation of object-oriented techniques in program structuring for the provision of design fault tolerance, guaranteeing that redundancy is incorporated in a disciplined manner so that the impact on system complexity can be kept under control. We show how object-oriented techniques can be used to build reusable components that support design fault tolerance. More specifically, we describe the implementation of reusable components that support the use of forward error recovery and software fault-tolerant techniques in a C++-like notation.

We adopt the terminology of Lee and Anderson[91], which has already been introduced in Section 3.1. Generally speaking, a system is viewed recursively as consisting of a set of components structured according to a design. A fault in a component may cause an error in the internal state of the system which eventually leads to the failure of the system. Forward error recovery techniques attempt to correct an erroneous state, while backward error recovery techniques restore a previous state which is presumed to be free from errors.

Idealised fault-tolerant components provide a coherent means by which the provisions for fault tolerance could be implemented in a system, in a way which minimises the impact of system complexity. These components receive requests for service and produce responses. If a component cannot satisfy a request for service, then it will return an exception. At each level of the system, an idealised fault-tolerant component will either deal with exceptional responses raised by components at a lower level or else propagate the exception to a higher level of the system.

In practice, we provide a collection of helpful abstractions for constructing software fault-tolerant components. It is required that such abstractions provide a clean separation between application functionality and software fault-tolerant mechanisms. Therefore, it is necessary to have powerful software composition mechanisms to build up these fault tolerant components. We believe that object-oriented techniques support such compositional approach via mechanisms such as behaviour sharing (including inheritance and delegation), generic functions, polymorphism and reflection.

The structure of the remainder of this discussion is as follows. First, we demon-

strate how inheritance can be combined with exception handling to implement a series of alternates implementing forward error recovery. Then we show how software fault tolerance may be implemented to mask possible failures of the components by means of three different object-oriented solutions. The first of these approaches concerns the implementation of a generic recovery block function while the second and third approaches explore a more generalised structure for implementing fault-tolerant components based on abstract base classes and polymorphism. Finally, the different solutions are evaluated and some problems and limitations are discussed.

3.3.1 Forward Error Recovery

In order to illustrate our approach for implementing forward error recovery, we take as an example a class implementing a collection of integers. First, we demonstrate how inheritance can be used to derive a more efficient implementation of the basic class collection. Secondly, we show how forward error recovery can be used to tolerate faults. Thirdly, we discuss how this approach can be generalised to build a hierarchy of idealised fault-tolerant components.

3.3.1.1 Motivating Example: Collection Class

Consider a class SimpleCollection representing a collection of integers with operations find, min and sort testing for the presence of a particular integer, returning the smallest integer in the collection, and sorting the collection of integers, respectively. Here is the base class definition:

```
class SimpleCollection
{
public:
        virtual boolean find(int); // linear search
        virtual int min(); // linear search
        virtual void sort();
private:
int  size;
        int *contents; // array
};
```

If no assumptions are made about the ordering of the elements in the collection, then the simplest implementation of find and min could use a linear search. How-

ever, if the array is always guaranteed to be in sorted order, a more efficient implementation can be derived from SimpleCollection using inheritance. The operations find and min are declared to be virtual functions so that they can then be redefined in a subclass of SimpleCollection.

Using the sort function, it is possible to implement a faster version of the original SimpleCollection class by trying to keep the contents of the collection sorted. For example, we could use a binary search for implementing find and return the zeroth element of the array for the min operation, if the collection is known to be sorted.

However, rather than keeping the collection sorted at all times, we choose to sort it when needed and allow it to become unsorted as a result of adding new elements. Thus, the fast implementations of find and min must first ensure that the array is sorted before they can exploit this property.

To implement this new class version, called FastCollection, we need to add an isSorted flag to the class. Every time the collection is updated by adding a new element, the flag is set to false to indicate that the collection is no longer sorted. However, if find or min is called, the array is sorted and remains sorted until the contents of the collection are modified again. An additional private member function ensureSorted is used to check the flag and sort the array if necessary:

```
class FastCollection : public SimpleCollection {
public:
  virtual boolean find(int);
  virtual int min();
private:
  void ensureSorted();
  boolean isSorted;
};
```

The implementation of ensureSorted and min for FastCollection is as follows:

```
void FastCollection::ensureSorted()
{
        if ( ! isSorted)
        {
                Sort(contents);
                isSorted = TRUE;
        }
        return;
```

```
}

int FastCollection::min()
{
        ensureSorted();
        return contents[0];
}
```

Similarly, the implementation of find for FastCollection also calls ensureSorted before using a binary search algorithm to determine whether the collection contains a particular integer. FastCollection implements a more efficient version of SimpleCollection but the correctness of FastCollection depends on the correctness of the sort function. In the next section, we show how to tolerate faults in the sort operation using forward error recovery techniques.

3.3.1.2 SafeCollection Class

If we make the assumption that the sort operation is unreliable, the implementation of the operations find and min provided by FastCollection will fail. However, provided the contents of the array have not been corrupted by the faulty sort operation, the implementations of find and min provided by SimpleCollection are still available and may be used to recover from the failure. Thus, the strategy for forward error recovery in this situation depends on being able to detect the failure of the sort operation in FastCollection and use the less efficient operations inherited from SimpleCollection to recover.

Assuming that the version of the sort operation used by FastCollection raises an exception in the event of a failure, a new class derived from FastCollection, called SafeCollection, is created. A SafeCollection is able to tolerate faults in the sort operation used to implement FastCollection by using the operations inherited from SimpleCollection as fallback positions. The class definition of SafeCollection is:

```
class SafeCollection : public FastCollection
{
public:
        virtual boolean find(int);
        virtual int     min();
};
```

A safe implementation of the min function looks like this:

```
int SafeCollection::min()
{
        try
        {
                return FastCollection::min();
        }
        catch (...)
        {
                return SimpleCollection::min();
        }
}
```

3.3.1.3 A Hierarchy of Idealised Fault-Tolerant Components

This structure can be generalised and used to implement a hierarchy of idealised fault-tolerant components within a class hierarchy. The implementation of each operation is separated into two distinct parts: a *normal* part and an *exception handling* part. The exception handling part either throws an exception or invokes a fallback implementation of the operation from a higher level in the class hierarchy. At each level of the class hierarchy, it would be possible to inherit or redefine both the normal and exception handling parts for each basic operation.

To summarise, inheritance is used to provide a series of fallback positions so as to implement a hierarchy of generalised fault-tolerant components, where each component redefines or inherits the normal and exceptional handling parts for each operation. In the next section we discuss the implementation of schemes for the inclusion of redundancy in a software system based on backward error recovery.

3.3.2 Software Fault Tolerance

The discussion of forward error recovery and idealised fault-tolerant components in the previous section illustrated one way of building reliable software. However, we assume that a complex software system will inevitably contain residual faults despite all of the fault prevention techniques which may have been used. So design fault tolerance is required if it is necessary to tolerate such faults, and its provision can only be achieved if design diversity has been anticipatedly built

into the system. Two main methods have been proposed for tolerating faults in software: recovery block and N-version programming. A simple generalisation of these two schemes is the following[8]:

(i) several variants are implemented for each software component,

(ii) each variant is designed independently, but should conform to the same interface specification,

(iii) a control framework (recovery block or N-version) executes the variants, and

(iv) the results are evaluated by an adjudicator.

Since the consequence of design faults are intrinsically unpredictable, backward error recovery is the most suitable error recovery technique for software fault tolerance. Backward error recovery techniques are general purpose because they make no specific assumptions about the errors in the system state. So in order to permit sequential execution of the variants, the recovery block scheme normally assumes the availability of backward error recovery, which involves the ability to preserve and restore object states.

Considering an object-oriented fashion for implementing such generalised components with diverse design, one could choose different levels for including software redundancy within the object model. We identify at least four categories according to the place where redundancy may be included, which we will refer to as *operation diversity*, *object diversity*, *class diversity* and *metaclass diversity*[163]. We address each of these categories below.

Operation Diversity. In this case, variants are operations/functions (not objects) declared in the class, which are independently developed from the same operation specification. The class designer may encapsulate the fault-tolerant operations within the class declaring them as private operations. An alternative choice is to declare the fault-tolerant operations explicitly in the public interface of the class. We consider this strategy to be an object-based solution for the inclusion of redundancy, and not an object-oriented approach.

Object Diversity. Redundancy is achieved by providing diversity in the data spaces of the classes. Fault tolerance would be achieved by instantiate a group of objects from the same parent class with diversity in their internal data, and then invoking the same operation on the group.

Class Diversity. Variants are objects and, therefore, both the state representation and the set of operations can be independently designed from the same specification of a type. Later we discuss two approaches for achieving such an abstraction in 3.3.2.2 and 3.3.2.3.

Metaclass Diversity. Redundancy could also be included at the metalevel depending on the application requirements. At the metalevel, it would be also possible to describe an implementation of redundant classes that could be used to build fault-tolerant objects at the application level.

In what follows, first we discuss the implementation of a generic recovery block function whose diversity is limited to the operation level. Then we show two approaches to introducing redundancy at the class level. Finally, we discuss some problems and limitations of the schemes presented.

3.3.2.1 Generic Recovery Block Function

A recovery block structure could be easily implemented as a generic function in C++. The template definition would be parameterised by a type T and the recovery block function itself would take four arguments: an object of type T, an array of alternates, the number of alternates, and an acceptance test. The alternates and the acceptance test would be pointers to functions taking an object of type T as parameter. For instance, the implementation of a fault tolerant **sort** operation could use a recovery block scheme to tolerate design faults in order to ensure the array order:

```
ensure array[i+1] >= array[i] for i = 1,..,N  // acceptance test
 by       sort array using quicksort          // quicksort variant
 else by  sort array using heapsort            // heapsort variant
 else error
```

Thus, given such a generic function for implementing recovery blocks, a robust **sort** operation could be implemented using **quicksort** and **heapsort** variants and a **sorted** adjudicator. The robust **sort** function catches any failure signalled by the generic recovery block function and throws a **SortException** instead.

Dealing with State Restoration

Backward error recovery techniques depend on restoring a previous error-free state by some mechanism. A very simple technique for implementing state based

recoverability would be to make a copy of the original object before updating it. One such approach is discussed in [42], where all "recoverable" objects are derived from a special class called LockManager which has two pure virtual function save_state and restore_state that each of its subclasses is required to define.

3.3.2.2 Variant and Adjudicator Classes

In this approach, which is discussed more fully in [163], variants are objects and a set of variants can be organised into different subclasses of the Variant abstract base class. All protocols for executing a variant are defined as *pure virtual* functions in the Variant class and inherited by its subclasses. The Variant class may provide a set of standard exception handlers which deal with some local errors detected during the execution of the user-defined variants. A basic set of adjudication algorithms would be provided by the system by means of an Adjudicator base class. The various standard control algorithms would be provided by the system through a class called SftComponent.

The construction of different variants to achieve design diversity can be very expensive. One advantage of using the approach here discussed for structuring an object-oriented system is that it allows the construction of variants by reusing existing components[128]. The inheritance mechanism allows a selective replacement or redefinement of a component's implementation parts. From this point of view, a subclass may be viewed as a variant of its parent class, and, therefore, it inherits code from the parent class. However, one must take care to ensure that such code reuse does not compromise design diversity by introducing too much commonality.

3.3.2.3 Generalised Object-Oriented Fault-Tolerant Components

In this approach service specifications are represented by abstract base classes and variants are instances of different subclasses that conform to the service specification of the abstract base class. Control frameworks and adjudicators are generic functions parameterised by the abstract data type representing the service.

More specifically, a control framework is a function that takes as parameters a set of variants, the name of the operation to be applied to each variant, and an adjudication function to evaluate the results generated by the variants. A library of control frameworks which implements different software fault-tolerant schemes

should be provided by the system.

To illustrate the approach let us take as an example the implementation of a robust stack. Figure 3.1 shows the **Stack** hierarchy which is now discussed. The **Stack** abstract base class defines the service specification by declaring the pure virtual operations **pop** and **push**. The C++ class definition for such a class would look like this:

```
class Stack
{
public:
   virtual int pop() = 0;           // pure virtual function
   virtual void push(int item) = 0; // pure virtual function
};
```

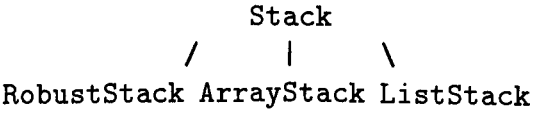


Figure 3.1: Stack Types

ArrayStack and **ListStack** are concrete classes derived from the **Stack** abstract class, which implement the variants for the service specification defined in **Stack**. A pure virtual function has no implementation in the base class and it must be overridden in each derived class. So a stack can concretely be instantiated in two different ways: in one case, to use an array representation (**ArrayStack** variant), and in another, to use a list representation (**ListStack** variant).

RobustStack is also derived from **Stack** and defines the visible part of the abstraction which will be used by the clients of **Stack**. The names of the variants are meant to be not visible to the clients, and access to objects of these classes is through references using virtual functions made by an **RobustStack** object. A **RobustStack** object is said to forward requests made of it to the variant objects encapsulated inside it. The set of classes together form an aggregate/composite object. Although there are multiple objects, the appearance to the user is that there is a single object - a **RobustStack** object - which manages the entire operation of the composite object. When a **RobustStack** object is instantiated, the variants objects **anArrayStack** and **aListStack** are also internally created. The implementation of the **RobustStack** class could be as follows:

```
class RobustStack: public Stack{
private:
  Stack *anArrayStack;
  Stack *aListStack;

public:
  RobustStack(){
       anArrayStack = new ArrayStack;
       aListStack   = new ListStack;}

 ~RobustStack(){
       delete anArrayStack; delete aListStack;}

  int pop();
  void push(int item);
  void voter();
}
```

A partial implementation of **pop** is schematically as follows:

```
int RobustStack::pop()
{
 ...
 NVersion({anArrayStack,aListStack},Stack::pop,voter);
 return item;
}
```

The implementation of **push** is similar. This abstraction of "outer and variants classes" can be interpreted from two different perspectives. On the one hand, the outer class can be viewed as delegating its functionality to collaborating variant objects. On the other hand, the variant objects can be thought of as being logically encapsulated inside the outer class, which enforces the notion of encapsulation. This approach can provide more polymorphism and runtime type support than inheritance using virtual functions alone.

3.3.3 Discussion

Reusable software components usually address functional requirements for a particular problem domain; however, here we have shown how object-oriented tech-

niques can be applied to implement reusable components which address non-functional requirements, in particular software fault tolerance, in the object-oriented class-based programming language. Since these components are built in an application-independent fashion, they can be reused across a wide range of possible problem domains.

We have shown how to use generic functions, inheritance and exception handling to implement forward error recovery and software fault tolerance in the form of reusable components that can be exploited by application programmers. Specifically, we discussed how inheritance can be used to provide a series of fall-back positions which in turn can be used to implement a hierarchy of idealised fault-tolerant components, each one redefining or inheriting the normal and abnormal case behaviour. We also presented a generic function that is used to implement the basic recovery block algorithm, and object-oriented approaches for the provision of software fault tolerance. Although the strategy of representing variants as objects (i.e. "class diversity") for the development of software fault-tolerant components seems to be the best choice, some problems arise, especially concerning with state saving and restoration. N-version programming requires an independent state for each variant, so it makes sense to represent a variant as an object. But the recovery block scheme uses a sequential composition, and the alternates are executed just when it is necessary so it does make much sense to keep information between calls. Some solutions for this problem are discussed in [163].

The implementations presented here have been based on basic object-oriented features, such as inheritance, dynamic binding and generic functions, likely to be found in many object-oriented programming languages, not only in C++. On the other hand, the Eiffel programming language does not provide support for generic functions. As a consequence, a possible Eiffel implementation of backward error recovery could be the creation of functions that behave like objects, that is, they have the role of a function but can be created, passed as parameters, and manipulated like objects.

Object-oriented techniques make it easy to represent the notion of abstract service interfaces supporting different implementations: just use an abstract base class to describe the interface and define a derived class for each kind of implementation. Moreover, the whole notion of an abstract data type supporting several different implementations associated with the notion of an outer object managing these different implementations is a fundamental way of keeping under control the complexity of real-world computer systems.

3.4 Object-Oriented Environmental Fault Tolerance

This section starts by motivating the need for a runtime association between an object and the properties usually associated with its class. The examination of the dynamical aspects of entities in the problem domain leads us to study the behaviour of an object over time. When we consider the behaviour of entities in the real world, we observe that these entities generally have a *lifetime*[137].

Since all instances of a class must follow the same rules of behaviour, when we abstract a group of similar entities (in the problem domain) into corresponding objects (in the solution domain) to build a class, we also abstract their common behaviour. Sometimes entities in the real world exhibit different phases of behaviour during their lifetime. For example, a butterfly begins life as a caterpillar, then changes into a chrysalis and, finally, transforms into an adult butterfly.

The term *logical state* is applied to each observable behaviour phase of the butterfly, that is, caterpillar, chrysalis, and adult butterfly. At any particular moment, different instances of a class may be in different logical state. The logical state that an instance is in is known as its *current logical state.* However, the behaviour implementation is different in these three distinct logical states. For example, an operation like *move* needs to be implemented in two different ways: one for creeping like a caterpillar and another for flying and walking like an adult butterfly.

In general logical states are information obtained at the design phase, and almost all current object-oriented language enforce that this information be buried in the implementation of the methods of an object. It is desirable to find means by which these logical states be represented very explicitly rather than implicitly through flag-checking code so as to obtain a better program structure.

The structure of the rest of this section is as follows. Section 3.4.1 introduces an example which is used to illustrate our ideas, and describes the concept of transmutable objects and their implementations using concepts such as delegation, abstract base classes and state hierarchies. Section 3.4.2 gives a more formal presentation of the delegation concept and Section 3.4.3 presents a technique for incrementally building the state hierarchy for a derived class from the sate hierarchy of its parent class based on the concepts of state machines and the restricted inheritance model. Following this, Section 3.4.4 discusses some related work. Although the example in this paper is written in C++[150], the ideas presented can also be applied with other object-oriented programming language, such as Eiffel[105]. Furthermore, the example is simple but was chosen merely to

facilitate the illustration of the ideas.

3.4.1 Motivating Example

Now we consider how these objects that have different behaviour phases could be implemented in C++[150]. To illustrate our ideas, we take as an example a simple checking account system, described in the following paragraph:

> A bank has two types of accounts: *saving* and *current*. Suppose that the bank is very strict regarding its current accounts. A *normal* current account must always maintain a positive balance, but, in return, the monthly charge is diminished. If it fails to do so, i.e., its balance drops to a negative number, the normal account is transferred into an *abnormal* account, and the monthly charge is augmented. If an abnormal account has a positive balance then it is automatically reverted to being a normal account.

3.4.1.1 First Implementation

The relationship between the account types of the bank is shown in the Figure 3.2. A possible solution to the problem is to embed the logical states of an CurrentAccount object within the attributes of the CurrentAccount class. Changes in the value of these attributes are detected by "if" statements buried within the methods of the object's class.

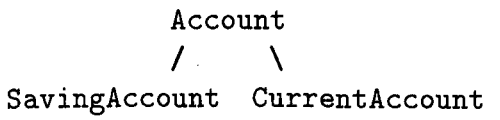


Figure 3.2: Account Types

The implementation of the CurrentAccount class could be as follows:

```
class CurrentAccount: public Account{
private:
  unsigned accountNumber;
  int      currentBalance;
```

```
  int      abnormalAccountFlag; // 1=abnormal 0=normal
public:
  CurrentAccount(accountNum, initialBalance);
  ~CurrentAccount();
  int balance();
  int debit(int amount); // process a check
  void credit(int amount); // process a deposit
  void dailyBalanceUpdate(); // update the account's balance at midnight
  void monthlyCharge(); // apply monthly charge
}

int CurrentAccount::balance() { return currentBalance;}

int CurrentAccount::debit(int amount) {
    if (currentBalance < 0)
       return 0;
    currentBalance -= amount;
    return 1;}

void CurrentAccount::credit(int amount) {
    currentBalance += amount;}

void CurrentAccount::dailyBalanceUpdate() {
     if (abnormalAccountFlag == 0) // normal account
        { if (currentBalance < 0)
             { abnormalAccountFlag = 1;}
        }
     else { // abnormal account
            if (currentBalance >= 0)
             { abnormalAccountFlag = 0;}
          }
}

void CurrentAccount::monthlyCharge() {
     if (abnormalAccountFlag == 0)
        { currentBalance -= MONTHLYCHARGE/2;}
     else { currentBalance -= MONTHLYCHARGE;}
}
```

The abnormalAccountFlag attribute indicates that the account is currently abnormal if its value is 1; otherwise, if its value is 0, it means that the account is normal.

An examination of this solution reveals that it has the same problems when compared to a typical non-object-oriented solution: the rules concerning logical state transitions of banking accounts are represented implicitly within the methods of CurrentAccount. The methods dailyBalanceUpdate() and monthlyCharge() that should only contain code from processing the monthly charge or changing the date at midnight should now also contain code to check and update the value of abnormalAccountFlag.

3.4.1.2 Second Implementation

The "if" statements buried in the methods dailyBalanceUpdate() and monthlyCharge() used in the first implementation could be eliminated. Since C++ is a language with runtime type resolution, the creation of "class types" like NormalAccount and AbnormalAccount would exempt the programmer to determine the type addressed by the abnormalAccountFlag. Thus, another solution is to expand the class hierarchy to include the description of the different types of current accounts (Figure 3.3).

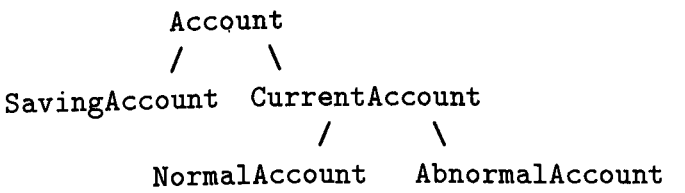


Figure 3.3: Expanded Hierarchy for Account Types

We might expand the CurrentAccount class as follows:

```
class CurrentAccount: public Account{
private:
  unsigned accountNumber;
  int      currentBalance;
public:
  CurrentAccount(accountNum, initialBalance);
  ~CurrentAccount();
  int balance(); // same implementation as before
  int debit(int amount); // same implementation as before
  void credit(int amount); // same implementation as before
  virtual void dailyBalanceUpdate();
```

```
  virtual void monthlyCharge();
}

class NormalAccount: public CurrentAccount{
public:
  void dailyBalanceUpdate(){
       if (currentBalance < 0)
           // account transition to AbnormalAccount
           // deleting a NormalAccount object and creating an
           // AbnormalAccount object}

  void monthlyCharge() {
        currentBalance -= MONTHLYCHARGE/2;}
}

class AbnormalAccount: public CurrentAccount{
public:
  void dailyBalanceUpdate(){
       if (currentBalance >= 0)
           // account transition to NormalAccount
           // deleting a AbnormalAccount object and creating an
           // NormalAccount object}

  void monthlyCharge() {
        currentBalance -= MONTHLYCHARGE;}
}
```

In this solution, the methods of CurrentAccount defined in the first implementation still exist for accessing information about an account. However, the methods dailyBalanceUpdate() and monthlyCharge() are now virtual and are overloaded in the NormalAccount and AbnormalAccount subclasses. For example, now the implementations of NormalAccount::monthlyCharge() and AbnormalAccount::monthlyCharge() take no extra action because the check for the value of a abnormal account flag is no longer necessary.

This solution now captures the relationship between the different types of current accounts. However, it has at least two limitations. First, class-based languages like C++, do not allow a program to change the class of its objects. When a current account changes state, for example, from NormalAccount to AbnormalAccount, a new AbnormalAccount object must be created and the old NormalAccount object deleted. However, this operation of deleting and creating objects could cause a large overhead for applications of reasonable size. Furthermore, it is not possible to the programmer to guarantee that the identity of the new object is identical

to the old deleted object. This is a crucial problem if other objects of the system contain references to the old object. A number of solutions have been suggested to solve such a limitation[40]. One solution is create a table containing pointers to all changeable objects. Another solution could be to keep a dependency list of all objects that have references to a changeable object. Thus, when the object changes its identity, it is possible to know all the objects that would require changing. Both solutions, however, add storage and performance overhead. What is needed is a mechanism that does not require objects to be deleted and recreated on every logical state change.

A second limitation is that NormalAccount and AbnormalAccount classes are not really subtypes/specialisations of an abstract current account, but they are separate conceptual states of the same abstraction. In fact, the logical states of a current account can be classified as normal and abnormal, and not the current account itself. All this discussion leads us to a third implementation that tries to solve the problems above mentioned.

3.4.1.3 Transmutable Objects

As discussed previously, it seems that two different issues have been confused in the second implementation: the class of an CurrentAccount object, and its current logical state. These two issues are represented by the same class hierarchy (Figure 3.3). However, one could choose to build two separate hierarchies (Figure 3.4), each representing a different issue:

- **Account Hierarchy** that represents the subtyping relationship between the different types of accounts, and
- **CurrentAccountState Hierarchy** that represents the subtyping relationship between the different logical states that an CurrentAccount object could progress through during its lifetime.

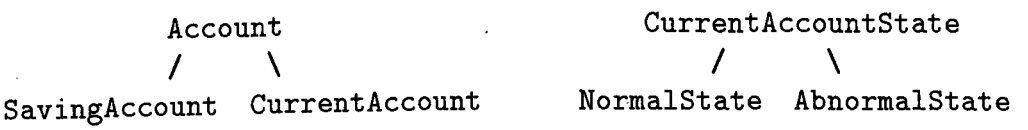


Figure 3.4: Parallel Class Hierarchies

A software design solution always has some underlying form or organisation, and to minimise complexity we need techniques for discovering this inherent form so that the design can be decomposed into a series of highly independent components[139]. So our proposed solution is to define a hierarchy of different subclasses related to an object, each subclass corresponding to a different implementation of its behaviour, and, then, to arrange that each object designates the responsibility for providing its operations to appropriate instances of these subclasses representing the different logical states that the external entity can exhibit. By this means it avoids having information about logical states hidden in the operations of the object.

Adopting such an approach, the C++ definition for the CurrentAccount class would like something like this:

```
class CurrentAccount: public Account{
      friend class CurrentAccountState;
private:
  unsigned             accountNumber;
  int                  currentBalance;
  CurrentAccountState *currentState; // the current state object
  NormalState         *normalState;
  AbnormalState       *abnormalState;
public:
  CurrentAccount(accountNum, initialBalance);
  ~CurrentAccount();
  int balance(); // same implementation
  int debit(int amount); // same implementation
  void credit(int amount); // same implementation
  void dailyBalanceUpdate(){ currentState->dailyBalanceUpdate(this);}
  void monthlyCharge(){ currentState->monthlyCharge(this);}
  void putNormalState() { currentState = normalState;}
  void putAbnormalState() { currentState = abnormalState;}
}

CurrentAccount::CurrentAccount(unsigned accountNum, int initialBalance)
{
  ...
  normalState = new NormalState;
  abnormalState = new AbnormalState;
  currentState = normalState;
}
```

A CurrentAccount object intercepts and forwards messages to currentState object, an "inner" state object, which represents the current logical state. The current state object gets dynamically rebound to the different logical state objects normalState and abnormalState as the CurrentAccount object changes its current logical state. In the example, the operations monthlyCharge() and dailyBalanceUpdate() are forwarded to currentState object for processing. The currentState object is updated through the state-changing method dailyBalanceUpdate.

The CurrentAccountState hierarchy is implemented as follows:

```
class CurrentAccountState{
public:
  CurrentAccountState();
  ~CurrentAccountState();
  virtual void dailyBalanceUpdate(CurrentAccount* ca)=0;//pure virtual
  virtual void monthlyCharge(CurrentAccount* ca)=0; //pure virtual
}

class NormalState: public CurrentAccountState{
public:
  void dailyBalanceUpdate(CurrentAccount* ca){
       if (ca->currentBalance >= 0)
            ca->putNormalState();}

  void monthlyCharge(CurrentAccount* ca){
        ca->currentBalance -= MONTHLYCHARGE/2;}
}

class AbnormalState: public CurrentAccountState{
public:
  void dailyBalanceUpdate(CurrentAccount* ca){
       if (ca->currentBalance < 0)
            ca->putAbnormalState();}

  void monthlyCharge(CurrentAccount* ca){
        ca->currentBalance -= MONTHLYCHARGE;}
}
```

In this implementation, the logical state of a CurrentAccount object can be changed without the need to delete and recreate the object. In addition, the programmer can add or modify logical states, and methods that control the transitions between them, without needing to change the methods related to the CurrentAccount

class. The classes that compose the CurrentAccountState hierarchy are referred to as *state classes*[45], and a CurrentAccount object, we term a *transmutable* object. So a state hierarchy represents sets of properties, each set describing a different logical state which implements basically the same interface specification. When messages are sent to a transmutable object, the interface implementation to be executed will be selected at runtime. The relationship between a transmutable object and its variant state objects can be thought of forming a *composite object*[156]. Although there are multiple objects, the appearance to the user is that there is a single object - the transmutable object - which orchestrates the entire operation of the composite object. When a transmutable object is instantiated, the state objects are also internally created.

The interface specification of the state hierarchy is defined by the abstract base class CurrentAccountState which declares the pure virtual operations dailyBalanceUpdate and monthlyCharge. AbnormalState and NormalState are concrete classes derived from CurrentAccountState which conform to the interface specification of the abstract base class. A pure virtual operation has no implementation in the base class and it must be overridden in each derived class. In other words, a group of related behaviour variants, that is, NormalState and AbnormalState, is defined as heirs of an abstract base class which defines a common interface specification, and as long as the transmutable object only access the behaviour variants via the abstract base class they are not affected when the current variant is replaced by a new one still conforming to the specification given by the abstract base class. Thus, specific reconfiguration changes can be easily incorporated to a software system for tolerating environmental faults.

Usually the state changes of an object over time, that is, its dynamic behaviour, is represented in a diagrammatic form known as a *state diagram*[131]. Such a diagram can be a basis for the construction of well-structured state hierarchies. Up to now, we have informally introduced the concept of *delegation*, that is, when an object forwards a requested message to some other designated object for processing. In the next section we define more precisely the concept of delegation, giving special attention to how delegation is implemented in a class-based object-oriented language like C++. Following that, we explore the construction of state hierarchies using state machines and the restricted inheritance model.

3.4.2 Delegation

There are many variations amongst existing object-oriented systems, especially concerning behaviour sharing and evolution. Behaviour sharing can be obtained

in two distinct ways: class-based sharing and instance-based sharing. All objects in a class have a common structure and share common behaviour. As a consequence, changes made to methods and structure in a class can automatically be passed on to all instances of that class. This implicit mechanism allows systems to be updated on a "per group" basis. In this sense, a class provides a template for all similar objects. Examples of class-based languages are C++[150], Eiffel[105, 106] and Simula[38].

On the other hand, the so-called delegation-based programming languages, such as Actor[2] and Self[154], adopt an alternative approach based on delegation[143], in which objects are viewed as *prototypes* (or *exemplars*) that delegate their behaviour to related objects, called *delegatees.* In this case, the system uses instance-based behaviour sharing instead of class-based behaviour sharing. Delegation-based systems can share both state and behaviour of objects. The *delegates-to* relationship can be established dynamically, while the inheritance relationship of class-based languages is established and fixed when a class is created. This is usually part of a system design that eliminates classes, focusing instead on concrete objects.

Three dimensions for behaviour sharing can be characterised in order to contrast class-based systems and delegation-based systems [55, 144]:

(i) static vs. dynamic - is sharing determined when an object is created or can it be determined dynamically?

(ii) implicit vs. explicit - are there explicit operations to indicate the sharing?

(iii) per group vs. per object - is sharing defined for whole groups of objects (classes) or could sharing be supported for individual objects?

Class-based languages use static, implicit and per-group strategies. By contrast, delegation-based programming languages use dynamic, explicit and per-object sharing strategies.

Evolution is concerned with how and when the links between nodes in a class hierarchy are allowed to change. In class-based systems, a method is invoked on an object as a result of sending that object a message. Usually the message includes: the name of the *receiver* and a *selector.* The selector carries the method name and appropriate arguments. The binding of the *method code* to the *method call* on the object is thus made at runtime. However, the pattern of sharing is *static* in that the runtime search for the method code will always traverse the superclasses in the same order.

In contrast, delegation-based systems have no classes, and methods can be stored in each object. An object can delegate messages to other objects, thus if the method lookup does not find the method name in the receiver, the search continues in the objects that the receiver delegates to, in the objects they delegate to, and so on. In this sense, an object "inherits" the methods of the objects to which it delegates messages.

Instance-based sharing can allow greater opportunity for implementing dynamic changes than class-based sharing. However, each scheme has advantages and drawbacks depending on the application domain. The security of compile-time class creation versus the flexibility of creating new objects "on the fly" is a central issue in the design of object-oriented systems. The security of compile-time class creation is a highly desirable characteristic in a language that is used for large static systems in which type security is needed. In this kind of system, to change the behaviour implementation of an object, the class is edited and the code passed again through the interpreter or compiler to recreate a new object that has the new specification. However, in class-based systems the requirement that each object belongs permanently to a particular class imposes constraints on the mutability of the behaviour implementation of an object. Sometimes some flexibility is desirable because many applications, such as process control systems or telecommunication switching systems, cannot be stopped to modify and extend their software[142]. It must be possible to replace some software components on the fly by new versions while the program is still running. The ability to modify and extend a system while it is running is known as *dynamic configuration*[86].

Evolution mechanisms in an object-oriented model are closely related to the binding between the method call and the method code. Essentially an evolution mechanism for a class-based system needs a runtime association between an object and the properties usually associated with its class to carry out its behaviour modification. However, class-based systems require that an object permanently belong to a class, and this can make the change of the behaviour implementation difficult. Therefore, a natural approach is to develop an evolution mechanism based on the delegation concept which allows the incorporation of changes over time, providing a great deal of programming flexibility.

3.4.2.1 Delegation in Class-Based Languages

Although inheritance and delegation are usually defined as alternatives in the design of an object-oriented system, delegation can be used in a class-based language as a way to implement behaviour sharing when an object needs, for example

because of an environmental fault, to be able to change its responses to messages at runtime. The main advantage that delegation has over inheritance is that delegation makes it easier for objects to change their behaviour implementation. Since it is dangerous to change the class of an object, most class-based languages do not allow it, but it is easy to change the delegatee of an object. Moreover, a language with static type checking can ensure that a delegatee will understand all the messages delegated to it. Thus, delegation is quite compatible with static type checking presuming that the delegatees of an object can be known at compile time. As a consequence, parent changing is limited to objects guaranteed to possess all the properties required by the descendants - there is no danger that behaviourally incompatible objects become parents as well. So delegation can be viewed as a design technique that can be used with any object-oriented language including those that are statically typed.

Delegation differs from simply sending a message in that the delegator continues to play the role of the receiver even after it delegates the message (as discussed in Section 2.2.4.3). In C++, delegation can be implemented by including the original receiver as an extra argument to each delegated message. An original message sets this argument to the receiver of the message, but delegated message sends do not change this argument.

Delegation-based languages implement this extra level of indirection automatically and invisibly. Hence one can easily implement behaviour changes of real-world entities by delegating operations to different objects representing the different kinds of behaviour that the entity can possess. But the fact that these different objects are intended to represent the same external entity at different logical states throughout its life time is not clearly captured by this implementation. It is also not possible to guarantee that all different objects representing the logical states of an entity conform to the same specification. Usually delegation-based languages emphasise flexibility, i.e., support for changes during runtime, relying on runtime type checking rather than static type checking so there is no guarantee that an object implements all the operations delegated to it.

We therefore explore a means of implementing delegation in a class-based language in order to improve the structuring of the program. We argue that the implementation of transmutable objects can be achieved in a more controlled and reliable way in a class-based language. We do not give up the fundamental principle that a language must be secure. However, sometimes it is necessary to have some support for changes over time when one embarks upon a real application.

The implementation of delegation in a class-based language requires more work

on the part of the programmer that it does in a delegation-based language. Extra work is required in defining the delegator and the delegatees. In Self, a method of a delegatee is automatically available to the delegator. In C++, this is not true. Thus, because delegation is being implemented "by hand", a method definition needs to be written in the delegator for each delegated operation. The function definitions are trivial, but this is an overhead for the programmer not present in a delegation-based language. However, a tool can be built to automatically generate code for the delegated operations. The operation in a delegatee class must have an extra argument to refer to the delegator. Instead of performing operations on *this*, the delegatee must perform operations on the delegator. There are two possible limitations with this ad-hoc way of implementing delegation. First, any class designed to be reused by inheritance must be modified before it can be reused by delegation. Second, classes reused by delegation are specialised only for that purpose.

3.4.3 State Hierarchies

The notion of state classes permit the specification of properties common to a group of states using a generalisation hierarchy, just as generalisation of states in Statecharts[65] allows the specification of common transitions. Statecharts also introduces the notion of aggregation of states. These two constructs provide additional power for modelling object-oriented systems. In this section, we provide some background about the use of state machines and their diagrams for representing classes, and show a technique for incrementally building the state diagram for a class from the state diagram of its parent class. State hierarchies can be represented as state diagrams. As a consequence, the same technique for incrementally building state diagrams can also be used for building state hierarchies, as we show in the next sections.

3.4.3.1 State Machines

A number of object-oriented design methodologies utilise a finite state machine to model the dynamic behaviour of objects, including the technique proposed by Rumbaugh *et al.*[131] and that of Shlaer and Mellor[137]. Rumbaugh's approach includes a *dynamic model* for a class that captures the changes in state undergone by instances in response to messages. Several styles of state diagrams have been reported on the literature[33, 58, 65]. Here we follow the notation introduced by Harel[65] for drawing (structured) state diagrams. The state hierarchy for the CurrentAccount class presented in Section 3.4.1.3 (Figure 3.4) can be represented

as a state diagram. Figure 3.5 illustrates the state diagram for the a current account. When **accountBalance** value is less than zero, the current account is ranked as abnormal. Similarly, when **accountBalance** has a positive value, the current account is considered to be normal.

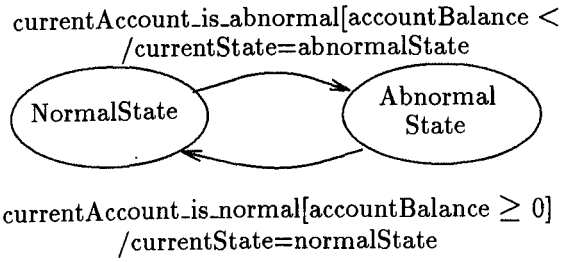


Figure 3.5: State Diagram for Current Account

Statecharts, originated by Harel[65], introduced the concept of hierarchy and aggregation into state representation. The ways of structuring state machines are similar to the ways of structuring objects: generalisation/specialization and aggregation/decomposition. Generalisation allows states and events to be arranged into generalisation hierarchies with inheritance of common structure and behaviour, similar to inheritance of attributes and operations in classes. More specifically, a *nested state diagram* is a form of generalisation on states, also known as the "or-relationship". An object in a state in the high-level diagram must be in exactly one state in the nested diagram. The states in the nested diagram are all refinements of the state in the high-level diagram. That is, one state may be divided into several independent subdivisions, only one of which describes the state of the system at runtime. For example, in Figure 3.6, **AbnormalState** is refined in two states: **OverdraftLimitExceeded** and **OverdraftLimitNotExceeded**. This combination of states is an exclusive-OR: to be in state **AbnormalState**, one must be either **OverdraftLimitExceeded** or **OverdraftLimitNotExceeded**, but not both. The state hierarchy derived from the description of this "or-relationship" is shown in Figure 3.7.

Aggregation allows a state to be broken into *orthogonal* components, with limited interaction among them, similar to an object aggregation hierarchy. Aggregation is equivalent to concurrency of states, also known as the "and-relationship". A state diagram for an aggregate is a collection of state diagrams, one for each component. For example, in Figure 3.8, **CurrentAccount** consists of AND components, **CurrentAccountState** and **CurrentAccountDebitRate**. Being in **CurrentAccount** implies being an aggregation of **CurrentAccountState** (either **NormalState** or **AbnormalState**) and **CurrentAccountDebitRate** (either **Student Overdraft** or **Unauthorised**

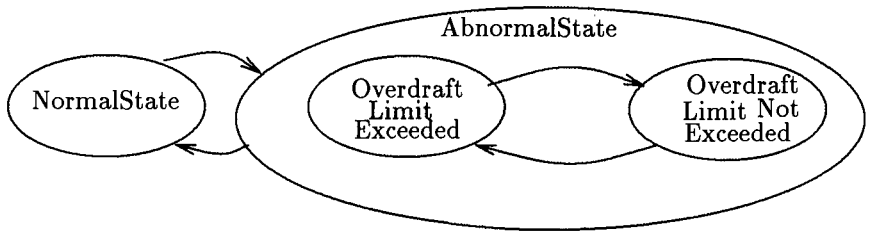


Figure 3.6: Generalisation of States for Abnormal State

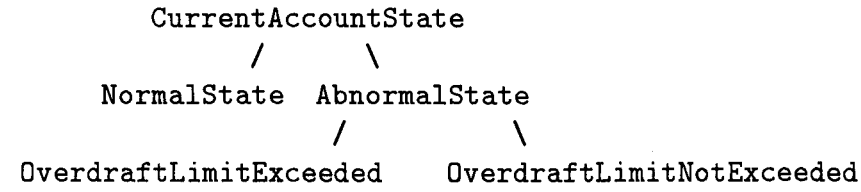


Figure 3.7: Expanded State Hierarchy for Current Account

Overdraft). In this example, there two kinds of debit rates which can be applied to a current account: *student overdraft* which corresponds to a rate of 0.52% and *unauthorised overdraft* which corresponds to a rate of 2.00%. CurrentAccount-DebitRate is a state orthogonal to CurrentAccountState so it is represented as an "and-relationship". The dashed line indicates two aggregate sets of states. The state hierarchy derived from the description of this "and-relationship" is shown in Figure 3.9.

3.4.3.2 Inheriting State Machines

The dynamic model of a class is inherited by its subclasses. The subclasses inherit both the states of the ancestor and the transitions, and they can have their own state diagrams. The assumption of the use of the restricted inheritance model allows us to deduce some implications about the state machine for a new derived class formed from a parent class[131][104]:

(i) *A derived class cannot delete a state of its parent class.* Given the assumption of the use of behavioural inheritance, the parent diagram must be a projection of the child diagram. It is possible to modify a state since the

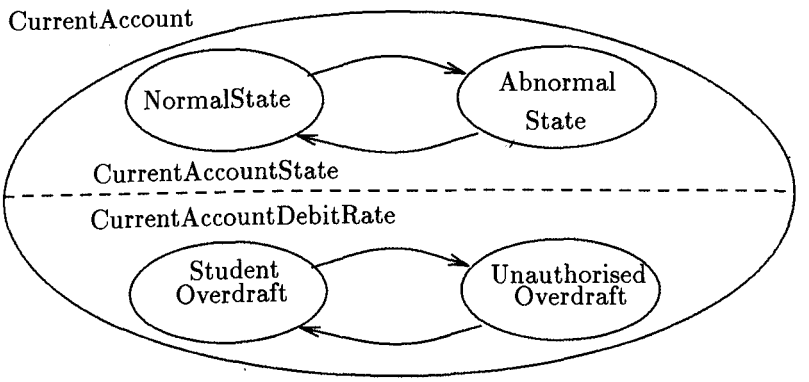


Figure 3.8: Aggregation of States

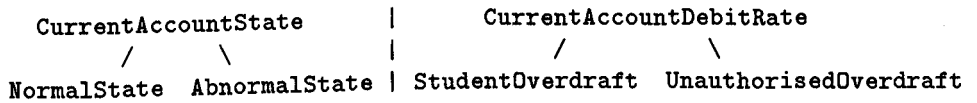


Figure 3.9: Parallel State Hierarchies for Current Account

method invariant still holds for every method in the class stated in the class invariant.

(ii) *Any new state introduced in a derived class is wholly contained in an existing state of its parent class.* The state diagram of a subclass may contain substates of an original state in the parent class (the "or-relationship"). Another possibility is the state diagram of a derived class be an aggregate/concurrent addition to the state diagram inherited from the parent class, defined on a different set of attributes, usually the ones added in the subclass (the "and-relationship"). A third scenario where the state diagram of the subclass would involve some of the same attributes as the state diagram of the superclasses generating a situation of conflict, is not possible under the assumption of the use of behavioural inheritance.

In order to exemplify the above statements, consider again the CurrentAccount class presented in Section 3.4.1.3 (Figure 3.4). The first example shows CurrentAccount class that has two states: *NormalState* and *AbnormalState*(Figure 3.10(a)). The subclass PrivilegedCurrentAccount is a special current account which offers a visa card feature. If the privileged current account is *normal*, the visa card feature is always available. However, if the privileged current account is *abnormal*,

the visa card can be automatically cancelled depending on whether or not the account's owner agrees to pay a fee of £5 per month. Thus the states *VisaCardCancelled* and *VisaCardNotCancelled* are not independent from the states of the CurrentAccount class. These new states are only meaningful if the current account is abnormal. Figure 3.10(b) shows this relationship where the states *VisaCardCancelled* and *VisaCardNotCancelled* are represented as substates of the *Abnormal* state ("or-relationship").

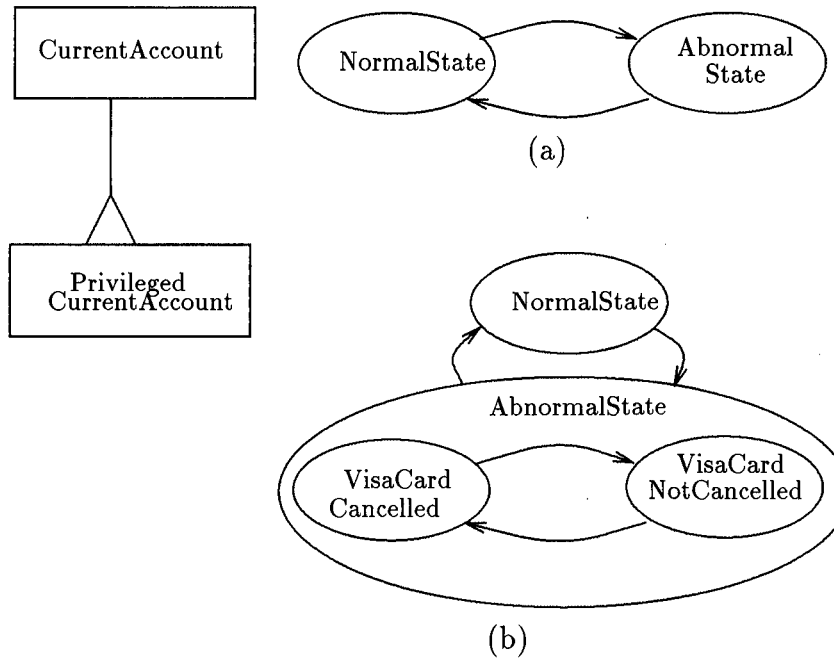


Figure 3.10: Example of Generalization of States

The second example is also based on the CurrentAccount class (Figure 3.11(a)). Suppose that SimpleCurrentAccount class, which is derived from CurrentAccount class, represents a current account in which service charges are applied. There are two kinds of services: *student* service which is free of transaction charges, and *graduate* service which is not. So one could create two states, for instance, *StudentService* and *GraduateService*, for representing this abstraction. The states *StudentService* and *GraduateService* are independent from the states of the current account ("and-relationship"). This results in the state diagram illustrated in Figure 3.11(b).

To summarise, this construction technique is mainly based on the concepts of hierarchy and aggregation in state diagrams. The first concept allows a state

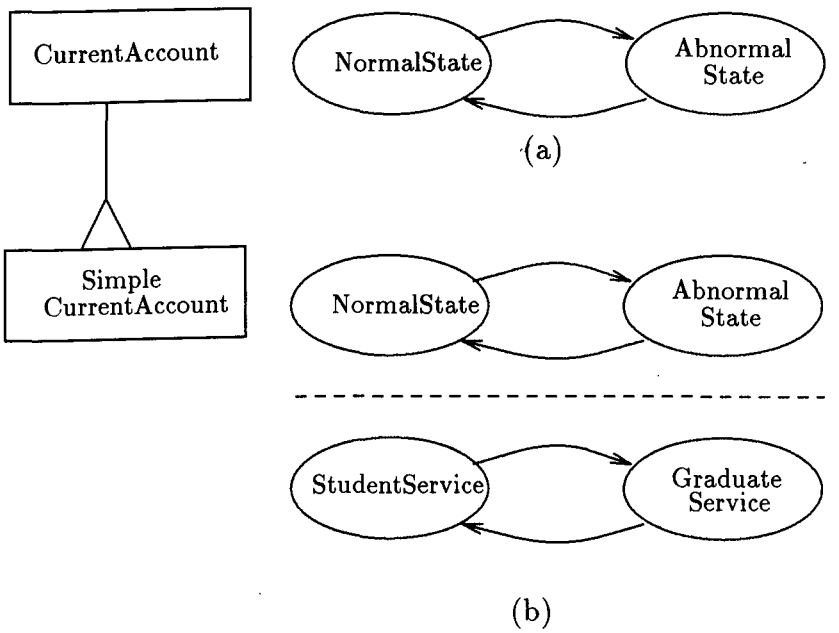


Figure 3.11: Example of Aggregation of States

from a parent class to be decomposed into two or more substates in the derived class. The second concept allows one to add a new set of states that are parallel to those that existed in the parent class. These two concepts together with the restricted inheritance mechanism provide sufficient power to incrementally modify the state machines from ancestor classes to form the machine that describes the derived class.

The series of implications derived from these assumptions (above described in (i) and (ii)) form a set of rules to be followed when designing such state machines. So the technique supports the reuse of state machine information along with the hierarchical structure constructed by the inheritance relationships among classes. States hierarchies can be represented as state machines. So the techniques for incrementally building state machines can also be applied to incrementally constructing state hierarchies for derived classes, based on the state hierarchy of their parent class.

Moreover, Harel's Statecharts are part of a larger development methodology that has been implemented as a commercial product called STATEMATE[67], which is a tool for the specification and analysis of complex reactive systems. One important feature of STATEMATE is the emphasis regarded the dynamic verification

of the behaviour specification: the tool provides facilities to program the model execution, in interactive or batch mode, and also to instrument the models to collect statistics during execution.

Statecharts is a graphical language which combines the visual appeal of graphics with rigorous mathematical semantics[66]. The semantics of Statecharts defines the sequence of system configurations taken into response to an external stimulus: starting from a *stable* configuration (i.e., no possible evolution unless an external event is generated), the system goes through a succession of *unstable* configurations until a stable configuration is reached. This chain of reaction is achieved by a broadcast mechanism: any internal modification can be sensed from all active orthogonal components, so that further transitions may be triggered. The broadcasted informations include the states exited or entered, the actions performed when the transition is taken (generation of events, modification of data items, etc.), the status modification of a controlled function (started, suspended, resumed, stopped), etc.

So the use of Statecharts for representing the behaviour of an object in terms of logical states, helps us not only with the construction of the state hierarchy but also with the formal verification of the transitions between the logical states. Moreover, this tool can also assist in the generation of automatic code for the implementation of transmutable objects.

3.4.4 Related Work

There are a number of different approaches to the provision of the sort of transmutable objects that are of relevance to our goal of providing a structured approach to environmental fault tolerance.

Dynamic Inheritance in Self

Self[154] is a prototype-base language with a simple and uniform object model. One consequence of Self's uniformity is that an object's parent slot, like other data slots, may be assigned new values at runtime. An assignment to a parent slot effectively changes an object's inheritance at runtime. Consequently, the object can inherit different methods and exhibit different behaviour implementations. This "dynamic inheritance" has been used is Self to implement objects with distinct behaviour modes. However, in general, parent changing is not limited only to objects guaranteed to possess all properties required by the descendents - behaviourally incompatible objects can also become parents.

Another related mechanism is the *become* operation found in Actor[2] and Self[154], which allows an object to change its interface, but it seems that this mechanism can destroy most of the benefits of type checking.

Modes

Mode[151] is an explicit language construct which extends the concept of ordinary class inheritance. In this approach, classes can have multiple sets of properties, each of which describes a separate conceptual state called *mode*. Each mode implements basically the same interface specification, and when an instance of a class is requested to perform some operation at runtime, the selection of the operation will be based on the set of operations in the mode in which the object is currently. In the case of defining a derived class from a parent class, because there are separate implementations of the same operations, the redefinition of methods in the subclass must be performed individually for each mode. Special transition functions can be defined to control the mode changing, so state change information is kept separate from operation implementations (each class can have only one transition function).

This approach has at least two limitations. Firstly, it does not allow the modes of a class to be organised as a state hierarchy. Secondly, when a new mode is added in a subclass is not sufficient to define only those operations for the new mode that have been incrementally defined in the subclass; rather, the new mode should be able to respond to all inherited operations as well (this requires a lot of work). Furthermore, the author states that it is not conceptually very clear how transition functions should be combined with inheritance in the case of adding new modes. This approach addresses many of the same issues as our approach although from a linguistic viewpoint. It should be mentioned that our solution tried not to create any new linguistic construction as a result.

Rôles in Fibonacci

Rôle is a concept from the Fibonacci database programming language[4]. This language has a notion of state and a concept of object identity; it also allows existing objects to be extended. For example, suppose that *John* is an object representing a person, when John enrols as a student the object might be extended to support a new behaviour, such as, a method *Number* that returns his student number. Let us call this extended object *JohnAsStudent*.

In Fibonacci, *John* and *JohnAsStudent* are considered to be the same object by the object identity test. Nevertheless, *John* and *JohnAsStudent* have different behaviours: the first does not understand the *Number* method while the second does. It is also possible, for instance, for John to take a part time job, requiring

that the object that represents him be extended in another direction, creating *JohnAsEmployee*. Moreover, *JohnAsEmployee* might also understand a *Number* method, which returns John's employee number. Fibonacci implements these abstractions by treating an object as a DAG of rôles; messages are sent to a receiving rôle, which first tries to "delegating" to subrôles, and then inheriting from superrôles. Thus the meaning of *John.Number* depends on the rôle type of *John*.

The rôle model emphasises the ability of an object to receive and send different messages at different stages of evolution, by means of changing dynamically the type of an object. Our approach restricts all the variants of an object to having the same basic interface whereas, in the rôle model, the interface of objects may well contain a completely distinct set of operations. Static type checking with state-dependent interface might require some interesting analysis similar to typestate checking in the Hermes language discussed below in this section.

Predicates Classes

Predicate class[29] is a linguistic approach which extends the standard object-oriented paradigm by reifying behaviour modes of objects. A predicate class has all the properties of a normal class, including a name, a set of superclasses, a set of methods, and a set of instance variables, and additionally, it has an associated predicate expression. A predicate class represents the subset of the instances of its superclass(es) that also satisfy the predicate. Whenever an object is an instance of the superclasses of the predicate class, and the predicate expression evaluates to true when invoked on the object, the object will automatically be considered to inherit from the predicate class as well. While the object inherits from a predicate class, it inherits all the methods and instances variables of the predicate class. If the object's state later changes and the predicate expression no longer evaluates true, the inheritance of the object will be revised to exclude the predicate class. Predicate class thus support a form of automatic, dynamic classification of objects, based on their runtime value, state, or other user-defined properties.

Although this approach is powerful and structured, it does not seem to provide a clean separation between the predicate classes and the normal classes; both kinds of classes are represented by the same hierarchy. The classes representing the predicates become easily confused with those representing subtyping. That is, the approach overloads the already rather overloaded inheritance mechanism. Again, this approach also addresses many of the same issues as transmutable objects, though from a linguistic viewpoint.

Hermes Typestate and Variants

The static type checking provided by the *predicate class* proposal earlier discussed in this section, ensures that a uniform interface is provided by an object, independent of its state at runtime. A less restrictive approach to type checking would allow different states of an object to have different interfaces similar to the *rôle* approach. For example, a **stack** object might have **empty_stack** and **non_empty_stack** state objects, but only the **non_empty_stack** object would support a **pop** operation. This approach would require a type state checking similar to the *typestate* checking in the Hermes language[146, 147]. Typestate is a refinement of the concept of type. Whereas the type of a data object determines the set of operations ever permitted on the object, typestate determines the subset of these operations which is permitted in a particular context. In this proposal, a type is not only characterised by the set of operations it provides but also by an automaton. The automaton associated to a type is defined as follows: states indicate the subset of the operations that may actually be applied, and transitions correspond to applications of the operations.

Hermes also supports the notion of *variant* type to weaken the constraint that the type of every value should be known at compile-time. A variant is just like a record, but whereas an initialised record may have several components, only one component of a variant can exist at any one time. Which component it is depends upon the case of the variant. For example, a Lisp object is either an *atom*, a *pair*, or nothing (*nil*). So the **LispObject** is a type whose case is an enumeration value, either *atom*, *pair* or *nil*. Each component is associated with one of the cases of the variant.

These notions have been introduced in the strongly-typed, concurrent object-oriented language called *Arche*[15], although the main concern was to characterise states under which a given method call may be selected. In this proposal, a type is extended by sets of peculiar type-states called *synchronisation-states*. Within a type, a synchronisation-state defines the method that may be executed by the objects of the type when they are to select an incoming message.

Other Related Work

Several other systems have constructs similar to aspects of transmutable objects. *Exemplars*[88] addresses the issue of forming automatic combination or union subclasses to avoid combinatorial explosion of classes and better organise methods. Many specification systems restrict the applicability of operations using preconditions and many concurrent systems allow operations to be conditional on guard expressions. Exception handling mechanisms share our goal of factoring cases, al-

though the main concern is to identify exceptional states. In this sense, one could represent exceptions as behaviour modes and use transitions functions as active preconditions: they can be programmed to capture errors at runtime and transfer objects to appropriate modes[151]. (This provides support for implementing an approach for forward error recovery.)

Johnson *et al.*'s approach[80] shares some of the characteristics of our approach, in particular with regard to the adoption of delegation as a mechanism for connecting the two separate hierarchies: one representing subtyping and the other representing the states of the object. The main focus of this work is delegation, and other issues, such as, the construction of state hierarchies in the presence of inheritance is neglected.

Another similar proposal is the *metaclass* approach[40]. According to the author, the two separate hierarchies indeed form a set of metaclasses that describes the states of an object. In our opinion, the two approaches are exactly the same, except for the terminology adopted.

3.4.4.1 Discussion

In my opinion, the most promising approaches described above for solving the problem proposed in this thesis are: predicate classes, rôles, and the Hermes language. Predicate classes provides the linguistic support for implementing the state hierarchies directly in the language. Moreover, the language offers support for controlling in which predicate class a given object is currently in. However, there are some drawbacks in this approach. The thing that I find most troublesome is that the same base class serves the root for the 'normal' classes and the predicates classes. That is, the class structure is not disjoint, generating a structure similar to the one discussed in Section 3.4.1.2, which contains not truly subtyping relationships.

The idea of rôles is also interesting, and we could represent state hierarchies using rôles. This approach is more general that predicate classes since an object can also change its interface. However, in such a case, it is not clear to me how the static typing checking is guaranteed. Finally, the concept of typestate seems to be well known, and it could be used to implement transmutable objects since the typestate determines the subset of operations applied in a particular context. However, Hermes does not offer support for representing the logical states in a hierarchical form. The notion of variant type is similar to 'variant record' in Pascal, but one cannot associate a set of operations to each component of the variant type. The advantage of using an object-oriented language is that support

for building hierarchies is automatically given.

Our approach has the advantage of implementability in any object-oriented language, however, the programmer has to follow a set of programming conventions. In response we could argue that it is possible to generate automatic code for transmutable objects with the use of a proper tool based on the object and dynamic models of a class. In our approach we decompose hierarchically the set of logical states associated with an object along with a controller which manages the transitions between the logical phases. The controlling object should not literally become one of the logical states, but it would intercept and/or delegate messages handled via an "inner" logical state that gets automatically rebound to different logical states as the changes are required. In this sense, the use of delegation is important because it provides an extra point of indirection. Also the use of polymorphism (more specifically inclusion polymorphism, refer to Section 2.2.3.3) is crucial in our approach; the inner state object is rebound to instances of the subclasses in the parallel hierarchy which are subtyping compatible with the inner state object (see Section 3.4.1.3).

Other polymorphic languages could be used to implement our approach such as POLY[68]. However, the advantages of using an object-oriented language become clear when we consider how a system might be extended. For example, if we already have '+' defined for integers and vectors, and we want to make it operate on recently added complex numbers, then in POLY we would have to modify the single routine which does adding, to make its body recognise the new value representation. In Smalltalk, for instance, we need only to define '+' in the set of operations of the number complex class; the uses of '+' for objects of other classes would remain as they are. As stated by Meyer[105], parametric and inclusion polymorphism are orthogonal, and ideally a language should support both. Languages such as POLY and ML, usually emphasise the use of parametric polymorphism only (refer to Section 2.2.3.3).

3.5 Conclusions

> "The ultimate object of design is form." [5]

The shift from an object-oriented conceptual model to an object-oriented implementation model is not always simple. Usually higher-level abstractions do not have a direct representation in an object-oriented programming language. One important consideration is where to place the complexity of these advanced abstractions. Some authors assert that an object-oriented programming language

should provide support for these notions directly[119], that is, their provision should be transparent for the object-oriented designer. The advantage of this solution is that putting the complexity in the language simplifies the development for the designer. However, overloading the language with too much things leads to a serious dilemma: the language itself must grow to express all alternatives.

An alternative approach is not extend the language, and provide building blocks (for example, classes and virtual functions) so that programmers could implement their own model of computation[34]. This approach leads to the creation of *idioms* using language features to express functionality outside the language, while giving the illusion of being part of it. It is asserted by Coplien[34] that experience has shown that these idioms can be effectively used without specific language support, giving the structure of programs more expressiveness, efficiency and aesthetic value. The abstractions discussed in this work are important examples of idioms that can be built using object-oriented building blocks. Concepts, such as delegation and runtime association between objects and classes, can be simulated in a strongly typed object-oriented language by means of extra levels of indirections and idiomatic constructions. The most important contribution of our work was to find an object-oriented approach for providing environmental fault tolerance without creating another new linguistic construction.

An approach for structuring behaviour phases - in particular faulty behaviour phases, in object-oriented systems has been proposed. Classes are very useful in structuring large systems. For real large applications, static typing checking is crucial. However, sometimes it is essential to have some support for performing changes during runtime. We presented an approach that benefits from the best of these two worlds: the security of compile-time class creation and the dynamic implementation of an object's behaviour via the delegation mechanism. The so-called transmutable objects can change the way that they respond to messages at runtime. This evolution mechanism is closely related to the binding between method call and method code. So when a transmutable object receives a message, the method look-up is done as in a class-based language. Once the method definition is found, the operation is delegated to the current state object, and then the method code is executed, depending on the logical state that the object is in.

The approach here described was applied to improve the structuring of a software controller with a complex and error-prone environment: a large model railway with 92 switches and 150 sensors, which is going to be explained in detail in the next chapter. The main goal of this application is to ensure that correct system operation is maintained despite the presence of environmental faults such as malfunctions of switches and sensors. Different behaviour phases (including normal

and abnormal ones) for switches and sensors are identified and a class hierarchy that captures these behaviour phases is developed in parallel with the application class hierarchy. The two hierarchies are linked together by a delegation mechanism - at any particular time, an application object will exhibit a particular behaviour implementation.

Moreover, in real-world application, not only environmental faults should be consider, but also hardware and fault tolerance should be provided (as we have already mentioned). We believe that approach described in Section 3.4 fits well with object-oriented structuring methods for supporting both hardware and software fault tolerance; however, this yet has to be demonstrated.

BLANK PAGE
IN
ORIGINAL

Chapter 4

Environmental Faults: A Detailed Case Study

> "Good design is possible to achieve, but it has to be one of the goals from the beginning."[118]
>
> "Bad design can rarely be overcome by more design, whether good or bad."[54]

The best way to understand the object-oriented techniques described in Chapters 2 and 3 is to use them in a realistic software system. So this chapter illustrates the use of the object-oriented techniques on a meaningful, complex application program. Many application experiences are being successful in applying object-oriented technology to an increasing number of different domains. It is very important to disseminate such information so that those interested in adopting the object-oriented approach can appreciate its applicability and build on previous success. However, many application projects fail to report on their experiences. Usually the projects concentrate too much on the system and its characteristics and lack any real technical substance on object-oriented technology. In fact, application experiences should be rich in technical details so that you gain a deeper understanding of the wide-ranging issues involved in development and its complexity, but, of course, the object-oriented issues should not get lost in this great deal of details. With such a view, this chapter aims to show the experiences of a project and then, based on these practices, generalise a set of useful insights/advices to provide an understanding of numerous issues, such as, reusability, design, structuring and maintainability.

This chapter focuses on the interest in object-oriented techniques in a particular community, namely, that working on fault tolerant systems. The approach described here is applied to improve the structuring of a software controller for a complex and error-prone environment: a large model railway with 92 switches and 150 sensors. Model railways are rarely very like real railways. For instance, the latter can be operated more dependably than the former, e.g., with the use of hardware redundancy in sensoring devices and actuators, such as, switches and signals. Moreover, in real railways, one can check whether commands sent to trains and actuators have been received or not; however in model railways this is not typical. Furthermore, in real railways drivers and signals play an important role whereas in model railways they do not exist. Human drivers are a potential source of failures; however, they might react better than a computer controlled system in critical situations.

Since the available model railway used to carry out our experiment is an unrepresentative model of a modern real railway, there is no intent to relate it to the art of railway control, rather the model railway is just being used as a system that has to be monitored and controlled. In our experiment with a model railway, the main goal is to ensure that correct system operation is maintained despite the presence of environmental faults such as malfunctions of switches and sensors. Different behaviour phases (including normal and abnormal ones) for switches and sensors are identified and a class hierarchy that captures these different phases is developed in parallel with the application class hierarchy. The two hierarchies are linked together by a delegation mechanism - at any particular time, an application object will exhibit a particular behaviour implementation. More generally, our approach allows separate class hierarchies to be linked together using delegation, with each hierarchy providing a different view of the same abstraction. By such means it is possible to specify different views of a system's behaviour, and the intended constraints on such behaviour.

The aim of this chapter is twofold. First, we will describe the methodology used for the design and implementation of the train controller, and show how the main concepts of object-oriented programming, such as inheritance, genericity and polymorphism, may be applied in practice. Second, we show how the software controller tolerates environmental faults which occur in the external process (i.e., the physical model railway) using a delegation mechanism. The rest of this chapter is organized as follows. Section 4.1 introduces the notation of the methodology adopted throughout this Chapter. Section 4.2 describes the Departmental Real Time System Environmental, that is, the model railway or simply the train set. Section 4.3 presents the problem statement for the train set system. Section 4.4 shows the analysis of the basic model for the application and Section 4.5 discusses its design and implementation. Following that,

Section 4.6 demonstrates how the basic model can be extended using delegation. Section 4.7 discusses some experiences in developing the train set system, and finally, Section 4.8 makes some concluding remarks.

4.1 Preliminary

4.1.1 Changes to the Traditional Lifecycle

The application of object-oriented concepts to software development brings new solutions to many issues, but also demands new methods and, in fact, a whole new approach. In a classical project, the phases of design, implementation and testing are generally viewed as separate global steps, to be executed in sequence on the whole system. In object-oriented programming, the approach is not the same. The sequence of design, implementation and testing applies to the lifecycle of individual classes but not necessarily to the lifecycle of the project as a whole. In other words, the design is not a distinct, monolithic phase; rather, it is just a step along the way in the iterative, incremental development of the system, whose steps may feed back each other.

In general, well-structured programs do more than simply satisfy their functional requirements. Programs that follow proper design guidelines are more likely to be correct, reusable, extensible, and quickly debugged. Most design guidelines that are intended for conventional programs (such as, partitioning, hierarchy, maximising module strength, etc.) also apply to object-oriented programs. In addition, features, such as inheritance and delegation, are peculiar to object-oriented programming and require new guidelines.

The methodology adopted for building the object-oriented model of the train set system is referred to as Object Modelling Technique (OMT)[131], which was summarised in Section 2.3.1. The next sections describe the object-oriented analysis for the train set application, following the main steps defined by this methodology, that is, problem statement, object model and dynamic model. Here we follow the notation introduced by Rumbaugh[131] for drawing object model diagrams, which is summarised below.

4.1.2 OMT Graphical Notation

Class

Figure 4.1 summarises the object modelling notation for classes. A class is represented by a box which may have up to three regions. The regions contain, from top to bottom: class name, list of attributes, and list of operations. Each attribute name may be followed by optional details such as type and default values. Each operation may be followed by an argument list and result type.

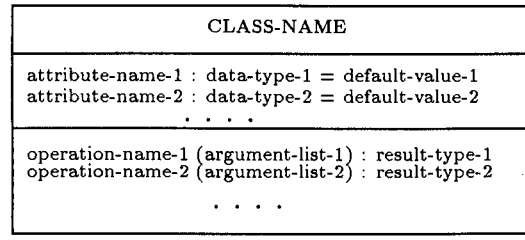


Figure 4.1: Class

Association

A *link* is a physical or conceptual connection between object instances. An *association* describes a group of links with common structure and common semantics. All links in an association connect objects from the same classes. An association describes a set of potential links in the same way that a class describes a set of potential objects. The notation used to represent an association is shown in the Figure 4.2.

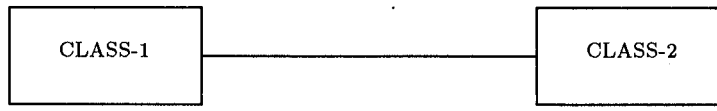


Figure 4.2: Association

Associations are inherently bidirectional. The multiplicity specifies how many instances of one class may relate to a single instance of an associated class. It constrains the number of related objects. Figure 4.3 summarises the multiplicity of associations.

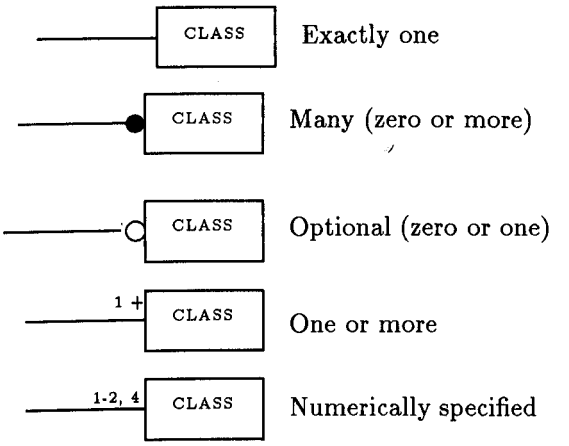


Figure 4.3: Multiplicity of Associations

Aggregation

Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the components of something are associated with an object representing the entire assembly. Aggregation is drawn like association, except a small diamond indicates the assembly end of the relationship. Figure 4.4 shows the notation for aggregation.

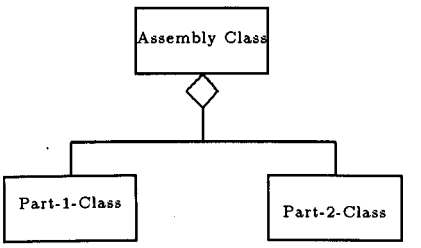


Figure 4.4: Aggregation

Generalization and Inheritance

Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass*. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to *inherit* the features of its superclass.

The notation for generalization is a triangle connecting a superclass to its subclasses. The superclass is connected by a line to the apex of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of the triangle, as shown in the Figure 4.5.

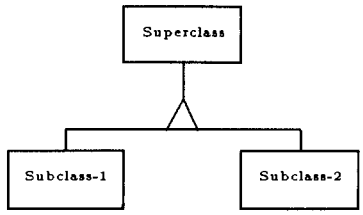


Figure 4.5: Generalization

4.2 The Train Set System

The train set is a digitally-controlled model railway, which is divided into three parts: electronic digital units, railway layout and trains. The railway layout is mounted on three separable boards which can be independently controlled by separate computers. These boards have red, green and blue labelled sensors, respectively. Each board can thus be viewed as being composed of connectors (e.g., switches), sensors and railway tracks (which link connectors and sensors) (see Figure 4.6).

Sensors are sensing devices for train detection and they are the only source of information about the state of the system. Experiments have shown that sensors are unreliable devices since they are sometimes erroneously triggered. In addition, switches are liable to suffer electro-mechanical faults, and consequently, trains can divert from a predefined route.

The Märklin Hardware

The powered units, for instance, engines and switches, are controlled using digitally encoded signals that travel through the track (via modulation of the electrical power supply of the trains). All messages go through the "central unit", which is connected to the signal path that runs along the middle of the track (Figure 4.7). The system allows up to a total of 256 switches to be controlled through digital signals. Figure 4.8 shows two different switches found in the train set. Each switch works on a double solenoid which enable the track to be set

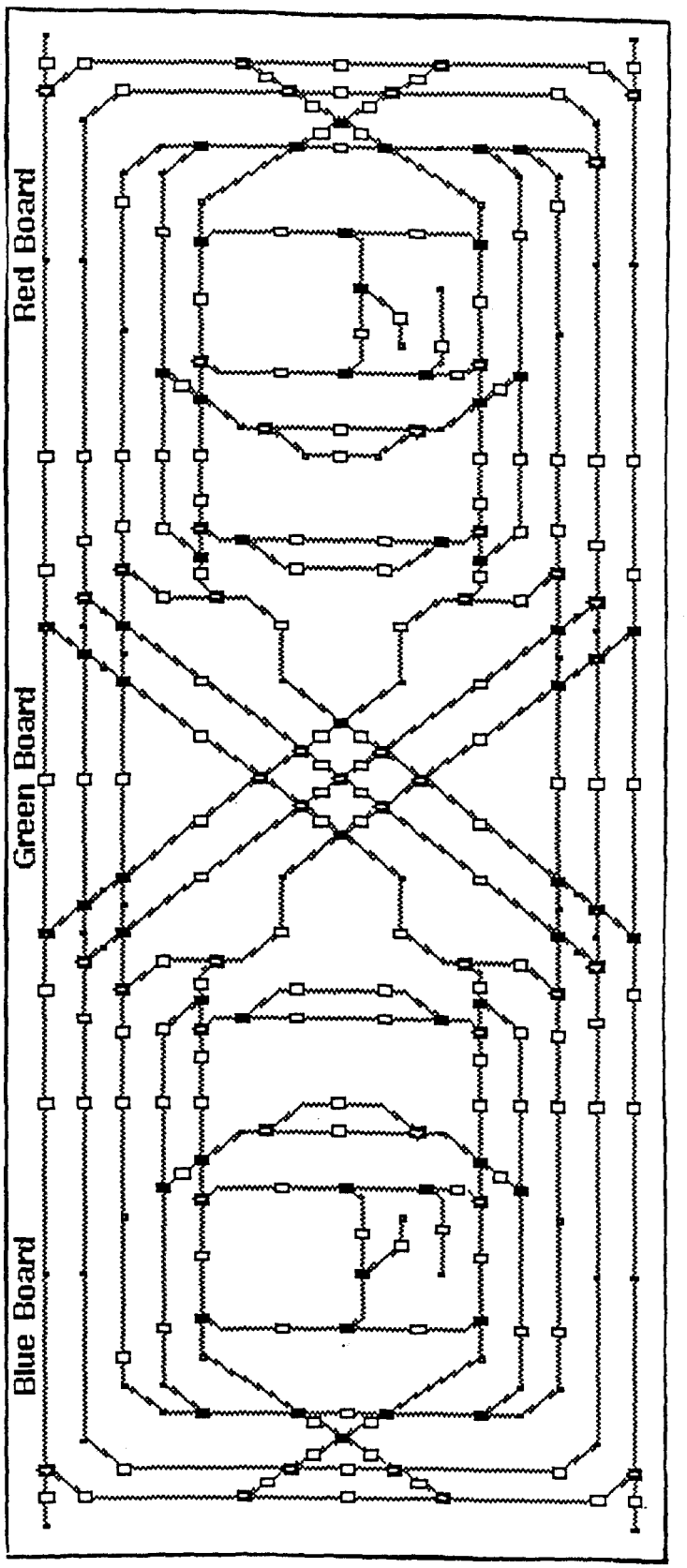


Figure 4.6: The Train Set Layout

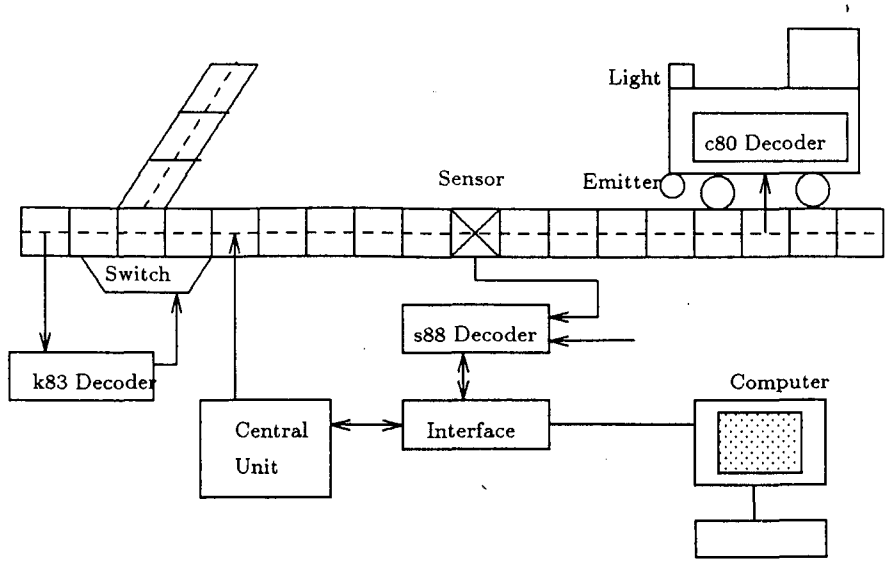
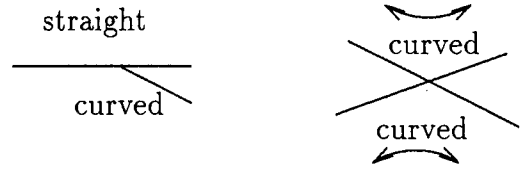


Figure 4.7: Diagram of the Märklin System

straight or curved. The solenoids are controlled through a Märklin k83 decoder. Each k83 decoder can operate up to 8 solenoids, i.e., 4 switches. Each decoder has a panel of 8 dip switches which is used to set up the addresses of the switches controlled by the decoder.



(a) Point Switch (b) Crossover Switch

Figure 4.8: Switch Settings

The engines have a bi-directional motor that is capable of running at 14 different speeds. Each engine has a c80 decoder, which has a bank of 8 dip switches via which the address of each engine is set between 1 and 80. All engines have a light function which shows the direction of travel. In addition, each engine is fitted with 2 infra-red light emitters, placed on the forward and backward ends of the engine. Track detection modules, represented by s88 decoders, work in conjunction with infra-red sensors to determine the engine's position on the layout. The infra-

emitters are mapped onto the light function so that only one of the emitters is activated, namely the one which shows the engine's direction of travel.

Infra-red sensors are fitted on various positions on the layout; all sensors are connected to s88 decoders. Each time an engine passes over a sensor, the sensor is triggered and this event is recorded in the s88 decoders. Each sensor is represented by one bit of information, and each s88 decoder can store up to 16 bits of information. The decoders are connected to the computer interface unit. On receiving a signal from the computer via the interface unit, the s88 decoder will dump the status of the 16 bits to the computer. The Märklin system can be operated from either manufacturer-supplied control units or from a computer via the interface unit. In our work, we are interested in the latter.

4.3 Problem Statement

We take for our application domain the train set and its object-oriented control program as our solution domain. Trains aim to move randomly between stations (that is, sensors, since they are the only source of information about the state of the system). Despite the presence of faulty switches and sensors, the trains should move around the railway without crashing, but if necessary stopping and reversing. Since the railway layout is divided up in three boards, the design should take into account both the layout distribution and the train crossings between neighbour boards.

The relevant restrictions to be taken into account when designing the train set application can be summarised in the following points:

- the aim is to guarantee no train collisions, i.e., safety.
- derailment of trains is not considered.
- switches can suffer environmental faults.
- sensors can also suffer environmental faults, but considering two consecutive sensors it is assumed that just one sensor can fail.
- routing and deadlock detection of trains is ignored.
- we assume that a train can stop within one section of the layout, that is, a train should travel slowly enough so as to hold this premise.

- we also assume that the length of a train is smaller than the smallest section in the entire layout, that is, a train can be completely contained within a section.

Functional Requirements

A variety of different projects can be envisaged that would make use of the train set, following on some projects that have so far been based on it[35, 9]. However it is desirable to avoid having each such project start unnecessarily from scratch - rather it would be better to have a standard interface to the train set which was as high as possible above the crude low-level interface provided by the Märklin hardware, but which was sufficiently low and well-chosen as to be appropriate for use in all, or at least the great majority of, future projects involving the trainset. (In practice there is likely to be a need for one or perhaps two interface levels below the "standard" interface to be made accessible - most particularly for (electronic and mechanical) hardware maintenance.)

The standard interface should in effect encapsulate three things: (i) the Märklin interface to the train set, (ii) the detailed representation of the train layout, and ideally (iii) a mimic diagram which accurately represents the train layout, and any relevant state information, e.g., about sensors, switch settings, etc. It should be capable of fronting either the entire three-board layout, or any of the separate boards. The detailed specification of this interface has been determined after examining the design of past train control projects and after discussions with prospective users of the train set. It was suggested that the principal objects made visible by the controller should be trains, sections and events. The controller should accept commands to move trains onto a specified next section, and should report back when it believes a train has actually done this, i.e., when the expected sensor has been triggered. (Sections are in effect directed links of railway tracks between adjacent sensors; an event is caused by a sensor being triggered.) The controller should also be capable of answering queries about what sections a given section leads directly to.

By such means the controller is intended to insulate the designers of director programs from needing to have detailed information about the physical layout of the train set, or of what track switches exist where - one consequence is that responsibility for deciding which of several possible different sets of track switch settings that would achieve the same link will remain with the controller. In other regards, the intent is that the controller be "policy-free", it being up to the operator how trains should first be located, where they should go, what precautions should be taken against collisions, what allowances made for possible hardware faults (in trains, sensors or switches), etc.

The controller should try to maintain an awareness of what train is on which link and hence is expected to trigger which sensor next (information that is needed for a mimic diagram), and to identify the train believed to be involved in any event when it reports this event to the operator. Equally it should inform the operator if an unexpected sensor is triggered - something that might occur spontaneously, or because an expected sensor triggering did not occur. The controller have responsibility for avoiding collisions. In addition, it also has responsibility for stopping trains running into buffers, that is, end points. (A end point is simply represented by the fact that a section has no next section.)

Ideally the controller should include a mimic diagram. The operator should be able to indicate whether it is using the entire three-board layout, or a individual board, and to switch the train set and/or the mimic diagram on (and off). Should just the mimic diagram be on, a simulated sequence of appropriate sensor events should be provided.

The software controller above described has been developed incrementally according to the requirements specified in Table 4.1. The main reason for adopting such an approach was to investigate how easily the model can be incrementally extended to cope with the new requirements. The basic prototype of the system (i.e., version 1) assumes that switches and sensors are reliable devices whereas the subsequent versions add gradually the new requirements. The aim was to introduce minimal changes to the basic model for the design of the following versions, so that the basic prototype would be completely reused. In the next section we carefully describe the object-oriented analysis for the basic model of our application. Following that, we will show the design and implementation of the basic prototype. After that, we extend the basic model to cope with the new requirements.

Requirements	*Version 1*	*Version 2*	*Version 3*	*Version 4*
Switch Fault Tolerance	no	yes	yes	yes
Sensor Fault Tolerance	no	no	yes	yes
Distributed Boards	no	no	no	yes

Table 4.1: Implementation of Requirements Through Versioning Cycle

4.4 Analysis of the Basic Model

Analysis is the first step towards solving the problem statement. The analysis addresses the three aspects of objects: static structure (Object Model), sequencing of interactions (Dynamic Model) and data transformations (Functional Model). All three models should be verified, iterated and refined. The methodology is not linear. Most analysis models require more than one pass to be completed. However, the three models are not equally important in every problem. Generally all problems have useful object models derived from real-world entities. Certainly the object model is the most important of the three models, since it describes the objects of the system, relationships between objects, and attributes and operations that characterize each class of objects.

In particular for the train set, the functional model is not essential since the application does not contain significant computation as compared, for example, to engineering calculations, which have a functional model with significant importance. Instead the dynamic model is more important than the functional model since the application concerns interactions and process control. So this section concentrates mainly on building the object model and the global event flow diagram (message exchange) for the train set application. The model described here provides the basis for the first version of the controller prototype. This model is further extended to accommodate new application requirements (mainly tolerance of environmental faults) in the subsequent versions of the prototype (described in Section 4.6). The remainder of this section is organised as follows. Section 4.4.1 describes the object model for the controller and Section 4.4.2 presents the message exchange diagram between classes.

4.4.1 Object Model

The first step in analysing the requirements of an application is to construct an object model. The object model shows the static structure of the real-world system and organises it into workable pieces. The object model describes real-world object classes and their relationship to each other. To construct an object model, first (i) we identify classes and their associations since they affect the overall structure of the system. Next, (ii) we add attributes to further describe the relationship of classes and their associations. Then (iii) we combine and organise classes using inheritance. Finally, (iv) we add operations to classes later on, based on the dynamic and functional models.

In the chosen methodology, the following steps should be followed for building an

object model:

1. identify object classes.
2. prepare a data dictionary containing descriptions of classes, attributes, and associations.
3. add associations between classes.
4. add attributes for objects and links.
5. organize and simplify object classes using inheritance.
6. test access paths using scenarios and iterate/refine the above steps as necessary.
7. group classes into modules, based on close coupling and related function.

4.4.1.1 Identifying Object Classes

Now, following the steps listed above, we extract objects from the requirements. Starting with the *tangible* entities[1] described in the requirements, we list candidate object classes:

`Train set system`	`Crossover`	`Switch`
`Board`	`Crossing`	`Operator interface`
`Sensor`	`Mimic diagram`	`Marklin interface`
`Train`	`Controller`	`Section`

Once we have defined the boundary of each object class, we should identify the topmost classes. Next, we make design decisions regarding the semantics of each of these abstractions, as well as, their relationships. From the highest level of abstraction, there is exactly one object which is the train set system itself. If we think about the ways in which we can manipulate this object from the outside, we could perform two meaningful operations, namely: powerDown and powerUp. These two operations thus form the essential protocol of the interface for the class TrainSetSystem.

[1]Tangible Objects are abstractions of the actual existence of some thing in the physical world.

There is a structural hierarchy within this top-level class: an object of the class TrainSetSystem is composed of a cooperating collection of other objects. In this kind of structural hierarchy, lower-level objects are hidden in the implementation of the enclosing object. It would be a bad design decision to make directly visible instances of all the object classes that we listed earlier. Such decision would increase complexity, since many of these objects are at different levels of abstractions. A better design decision is to review the list of object classes and select only those which correspond to the elements at the highest level of abstractions. As a consequence, we have decomposed the object model for the train set system into four modules: train set, controller, board and train, which are described in detail in the following sections.

4.4.1.2 Train Set Module

Figure 4.9 illustrates the top-level object model diagram. The class TrainSetSystem has a single instance that has an operator interface and a controller. Thus, we show that the class TrainSetSystem is composed of the classes OperatorInterface and Controller. Figure 4.9 also shows the cardinality of the associations: there is exactly one operator interface and controller. Similarly, each operator interface and controller belongs to the same train set system.

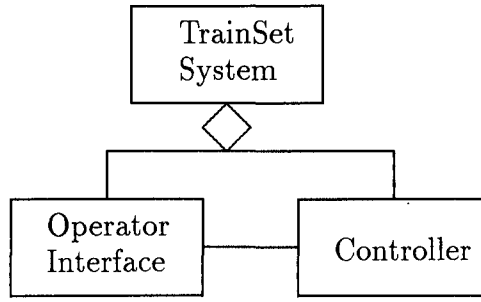


Figure 4.9: Train Set Module

The class Controller should pass messages to the class OperatorInterface, and the class OperatorInterface should pass messages to the Controller instance. Figure 4.9 captures this design decision clearly; it shows that the pair of these classes is associated with each other. For instance, the operator create trains, specify how trains should be first located, where they should go, what precautions should be taken against collisions, what allowances made for possible environmental faults (e.g., sensors and connectors) and when to start and stop the system. On the

other hand, the controller should report to the operator the current train position, and also inform the operator if something exceptional occur, e.g., if an unexpected sensor is triggered or if an expected sensor triggering did not occur.

4.4.1.3 Controller Module

Figure 4.10 shows the controller module. The class **Controller** has a single instance which encapsulates the Märklin interface, the physical layout of the train set (i.e., the board), the mimic diagram and trains. In other words, the controller is the central part of the system, which is broken down into many control objects, such as, train controllers (one for each train) and board controller. Thus, we show that the class **Controller** is associated with the classes **MarklinInterface**, **Board**, **Mimic** and **Train**. The class **MarklinInterface** represents the low-level interface provided by the Märklin hardware. The class **Mimic** shows the train layout and any relevant state information, e.g., about sensors and switch settings. The class **Board** defines the detailed representation of the train layout. The class **Train** represents trains moving around the board.

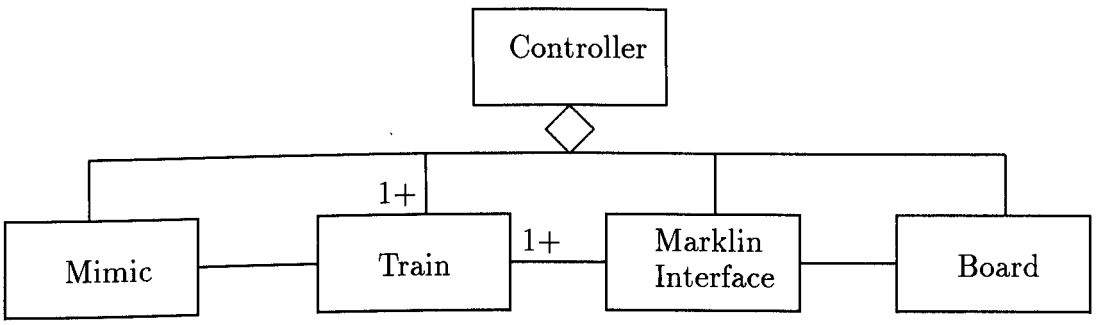


Figure 4.10: Controller Module

More specifically, the controller has exactly one Märklin interface, board and mimic diagram, and a number of train controllers. Similarly, each Märklin interface, train, board and mimic diagram belong to the same controller. The class **Controller** should pass messages to objects of the classes **MarklinInterface**, **Board**, **Mimic** and **Train**, and objects of the classes **MarklinInterface**, **Board**, **Mimic** and **Train** should also pass messages to the **Controller** instance. Therefore, the class **Controller** should be mutually visible to the classes **MarklinInterface**, **Board**, **Mimic** and **Train**. In a similar fashion, each pair of classes **Mimic** and **Train**, **Train** and **MarklinInterface**, **Board** and **MarklinInterface** are mutually visible to each other.

For example, the implementation of the class Train should see the interface of the classes Controller, MarklinInterface and Mimic. A train passes messages such as lockSection and releaseSection, which are defined in the class Controller. A train also passes messages such as updateTrainPosition, which is defined in the class Mimic. A train also send messages such as setSpeed to change its speed, which is defined in the class MarklinInterface. In a similar way, the class Board passes messages such as setDirection, which is defined in the class MarklinInterface, in order to change the direction of a switch.

4.4.1.4 Board Module

Figure 4.11 shows the board object model diagram with associations. The class Board is an aggregation of its sections, which are in turn aggregations of their stations (that is, sensors) and connectors (e.g., switches). As mentioned earlier, a *section* is a directed link of railway tracks between two adjacent stations. Each section, delimited by two adjacent stations, can contain a sequence of zero or more connectors (Figure 4.12). Connectors and stations are linked to each other through pieces of railway tracks which are called *edges*. The direction of a section s is defined as being from its tail station t to its head station h. For instance, in Figure 4.12 the section defined by the two adjacents stations A and B is different from the section that links B to A. Thus, a section is associated with its opposite section. The one-to-one association *oppositeSection* in Figure 4.11 shows this relationship.

An important notion related to connectors is their guiding points. A *guiding point* is an imaginary point near a connector tip. There are three types of connectors: crossing, endpoint and switch. (Figure 4.13). A *crossing* is a static kind of connector which cannot be controlled. An *end point* is a terminal connector of in the board. A *switch* is a connector that has two controllable directions: *straight* and *curved*. There are two kinds of switches: *point* and *crossover*. A point switch has three guiding points, while a crossover switch has four guiding points. A crossing has also four guiding points whereas an end point has just one. Each guiding point of a connector is connected to an edge. This is represented by the association between the class Connector and Edge in Figure 4.11; each connector is associated with one to four edges, conversely, each edge can be associated with zero to two connectors.

Station is an infra-red sensor used for train location. In the actual board there are some sensors which are located at the middle of connectors. However, such sensors are not eligible to be stations since they cannot determine which path

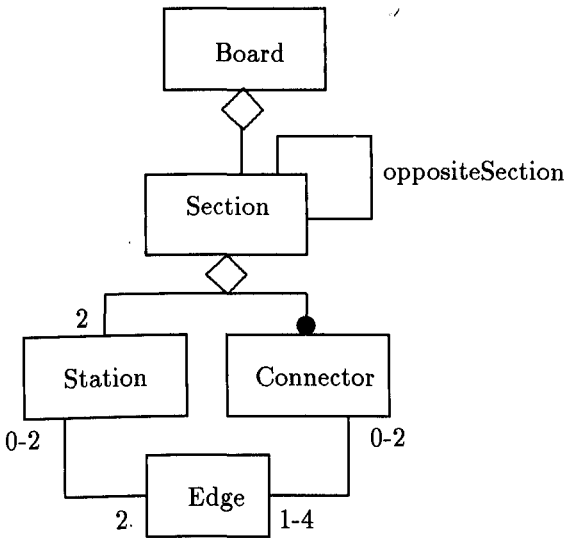


Figure 4.11: Board Object Model with Associations

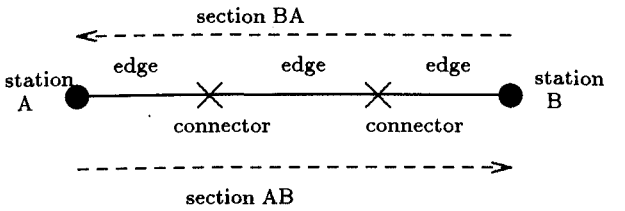


Figure 4.12: Example of a Section

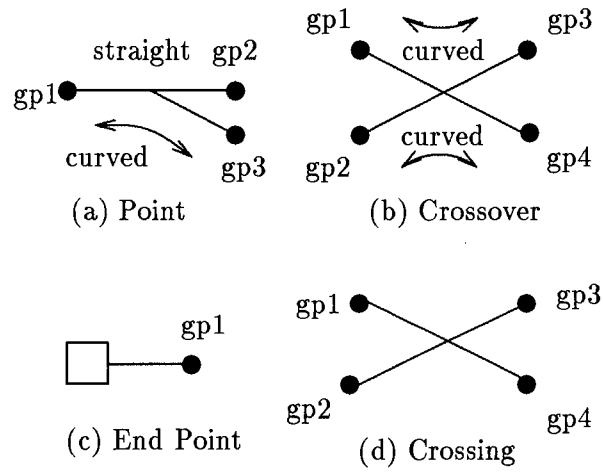


Figure 4.13: Kinds of Connectors

was taken by the train. The blue board contains 48 sensors, the green board contains 64 sensors and the red board has 48 sensors. All these sensors can be identified by labels in the layout. The signal of a sensor should be represented by "0" when a train has passed over it; otherwise it should be represented by "1". Experiments have shown that a sensor may be erroneously triggered, so a read-in "0" signal does not always represent a real detection signal. Each sensor is always associated with two edges. This is represented by the association between the class Station and Edge in Figure 4.11; conversely, each edge can be associated with zero to two stations.

Figure 4.14 shows the board object model diagram that represents the board abstraction. Connectors are classified into several different subclasses: EndPoint, Crossing and Switch. The subclass Switch, in turn, can be classified into different subclasses: Point and Crossover. We group the classes Connector and Station under the abstract superclass Vertex. More generally, a edge connects two adjacent vertices, which can be either a station or a connector. We also group the classes Edge, Vertex and Section under the abstract superclass BoardThing, which represents all the elements that compose a board.

As mentioned earlier, the controller is responsible for answering queries about what sections a given section leads directly to. The next sections of a section is an important notion concerning the train set structure. Given a section s with a tail station t and a head station h, the next sections of s are defined as the sections whose tail station is h. For instance, Figure 4.15 illustrates a

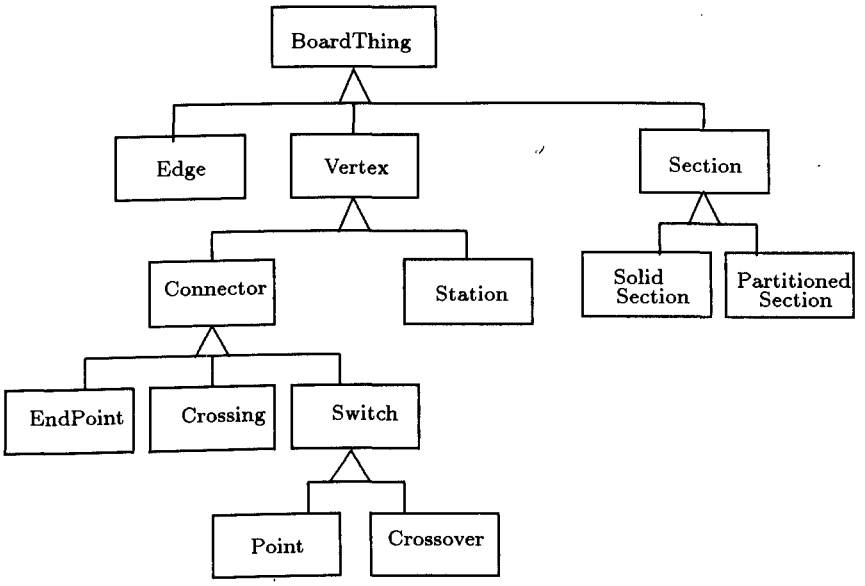


Figure 4.14: Board Object Model with Inheritance

portion of the physical layout topology, which contains four stations *stationA*, *stationB*, *stationC* and *stationD*, one switch crossover called *crossover1*, and two switch points labelled *point2* and *point3*). We can identify the sections described in Table 4.2. For instance, the next sections of the section currently occupied by the train (whose head station is *stationA*) are *section1*, *section3* and *section5*. The opposite sections of *section1*, *section3* and *section5* are, respectively, *section2*, *section4* and *section6*.

Section	*tailStation*	*headStation*	*path*
section1	stationA	stationB	crossover1 (straight), point3 (straight)
section2	stationB	stationA	point3 (straight), crossover1 (straight)
section3	stationA	stationD	crossover1 (curved), point2 (curved)
section4	stationD	stationA	point2 (curved), crossover1 (curved)
section5	stationA	stationC	crossover1 (curved), point2 (straight)
section6	stationC	stationA	point2 (straight), crossover1 (curved)

Table 4.2: Examples of Sections

After this explanation, we can identify two different kinds of sections in the board, namely, solid and partitioned sections. A *solid* section has next sections while a *partitioned* section does not have a next section. Figure 4.16 shows an example

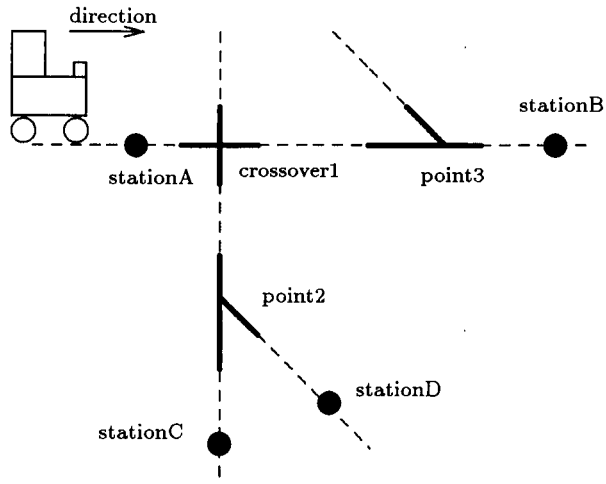


Figure 4.15: Example of Next Sections of a Section

of a typical partitioned section found in the train set layout. As a consequence, when the train occupies a partitioned section, the controller is responsible for stopping and reversing the train before hitting the end point. Thus, sections are also classified into two different subclasses: SolidSection and PartitionedSection, as shown in Figure 4.14.

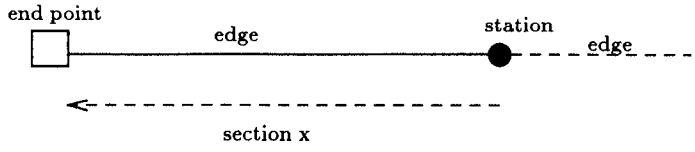


Figure 4.16: Example of a Partitioned Section

Figure 4.17 describes the final object diagram model for the board module with associations, inheritance, attributes and operations, which is a union of the figures 4.14 and 4.11. The class Section defines the attribute access that shows if a section is free, locked or occupied, a pair of adjacent stations defined by the attributes headStation and tailStation, the set of connectors settings that defines the section represented by the attribute path, and, finally, a pointer to a section object which defines its opposite section (oppositeSection). The class Section also defines the operations to lock (lock), release (release) and occupy (occupy), which change the access value of a section object.

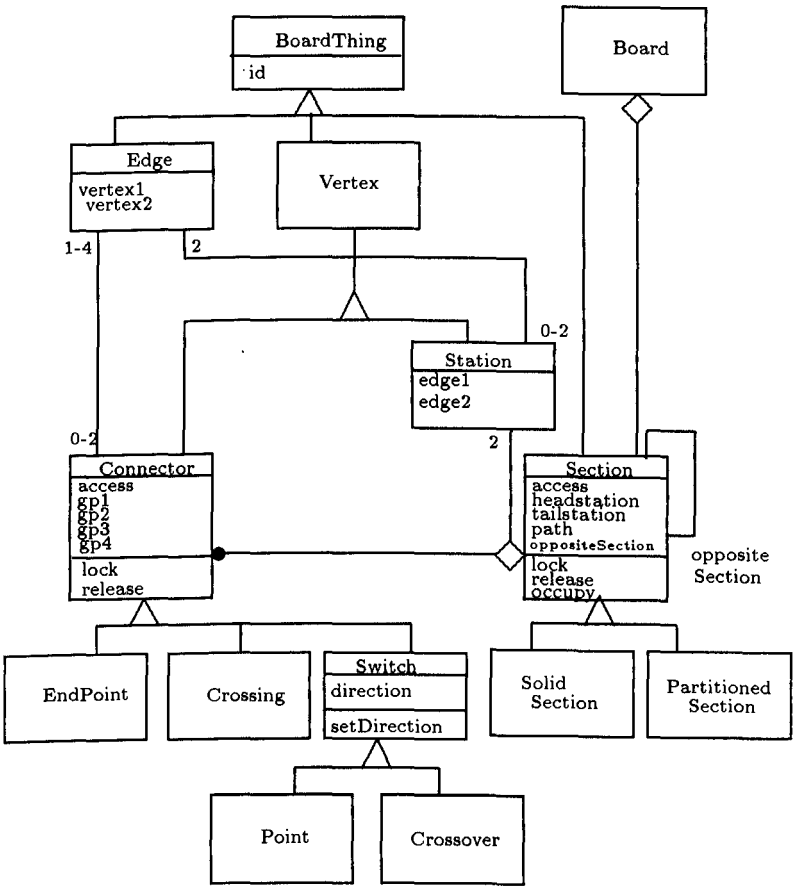


Figure 4.17: Board Module with Associations, Inheritance, and Some Attributes and Operations

The class Connector defines the attribute access which indicates if a connector is free or locked, as well as operations lock and release to change the access value. A connector also contains guiding points defined by the attributes *gp1*, *gp2*, *gp3* and *gp4*, which represent the edges. Depending on the kind of connector, the number of guiding points can be 1, 3 or 4 (see Figure 4.13). The class Switch is a kind of Connector, which has a direction straight or curved, as specified by the attribute direction, as well as an operation setDirection to change the direction value.

The class Station defines two edges edge1 and edge2 which a station object is connected to. In a similar way, the class Edge defines two vertices vertex1 and vertex2 which a edge is connected to. All the entities of the board have an identification, as specified by the attribute id defined by the class BoardThing.

4.4.1.5 Train Module

Figure 4.18 illustrates the train module. The class Train has many instances, each one representing a train moving around in the board. The class SensorPoller has a single instance, which is visible for all train objects, as specified by the one-to-many association between SensorPoller and Train. The class SensorPoller reports to a given train the event caused by an expected sensor being triggered. There is also a one-to-one association between SensorPoller and MarklinInterface. The class SensorPoller should pass messages to the MarklinInterface instance, such as readSensor and readAllSensors, which are defined in the class MarklinInterface.

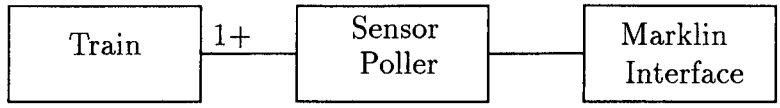


Figure 4.18: Train Module

As far as the safety of the train set system is concerned, the notion of control zone of a train is very important for avoiding train collisions. The *control zone* of a train is defined as being the front region acquired by a train, i.e., all sections locked ahead of the train. Each train has a control zone, and it is responsible for setting its route within its control zone. So a train knows what it is the next station to be triggered. When an unexpected sensor is triggered, the train will signal an exception.

We associate with control zone the notion of *levels* (see Figure 4.19). The first level holds information of the next sections of a given section. The second level

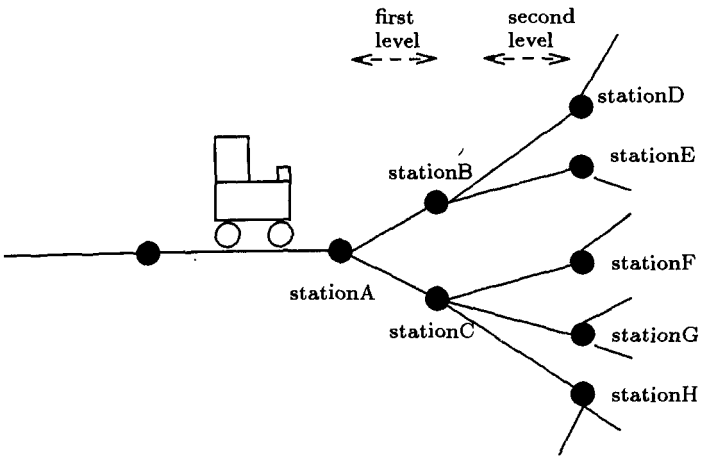


Figure 4.19: Example of a Control Zone with Two Levels

holds information of the next sections of each section of the first level, and so on. If we assume that sensors and switches are reliable devices, as it is the case of the basic model, it suffices to build a one-level control zone. Moreover, it is not necessary to acquire all the next sections of the current section of a train; it suffices to lock just one front section, chosen randomly by the train. Since switch and sensor faults are supposed not to occur, the control zone does not need to be enlarged to encompass all possible deviations of the route.

So each train before it starts moving should build a proper control zone, according to the assumptions made. This notion of control zone plays an important role in the detection and recovery of environmental faults by the train, and we will return to it again in Section 4.6.

4.4.2 Dynamic Model

The dynamic model specifies allowable sequences of changes to objects from the object model. The dynamic model represents control information: the sequences of events, states, and operations that occur within a system. A state diagram describes all or part of the behaviour of one object of a given classes. Usually events can be represented as operations on the object model. In summary, we perform the following steps in constructing an dynamic model:

1. prepare scenarios of typical interaction sequences.

2. identify events between objects and prepare an event trace for each scenario.
3. prepare an event flow diagram for the system (message change).
4. build a state diagram for each class that has an important dynamic behavior.
5. check for consistency and completeness of events shared among the state diagrams.

We begin by looking for events which are externally-visible stimuli and responses. If necessary, then we summarise permissible event sequences for each object with a state diagram. In our case, we concentrate on producing events flow diagrams between the various classes of the object model, since the classes have a relatively straightforward dynamic behaviour. Figure 4.20 summarises the main events between classes, without regard for sequence.

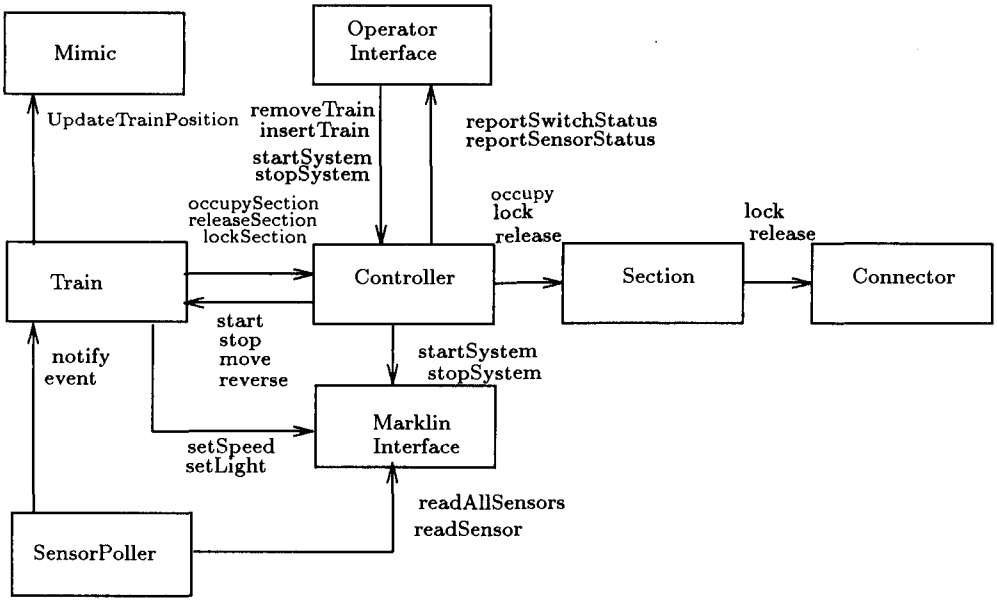


Figure 4.20: Event Flow Diagram for the Train Set System

4.4.3 Iterating the Analysis

The goal of the analysis is to fully specify the problem and problem domain without introducing any detail of a specific implementation. In practice, however, it

is sometimes difficult to avoid all details of implementation. Usually the analysis model requires more than one pass to complete, and this was the case for the train set. The train set analysis presented above was iterative and the model was refined many times as our understanding of the problem increased.

Along the route of the analysis, many inconsistencies and imbalances were found in the models. However, the iterations of the different stages have produced a cleaner, more coherent design. Moreover, some alternate designs were tried and evaluated. For instance, our initial idea was to define lockable blocks of tracks instead of sections. However, this idea turned out to be inefficient. For example, considering the portion of the green board illustrated in Figure 4.21, if a train acquires the whole block in the first place but its route uses just one of the straight lines, then we have allocated too much resources.

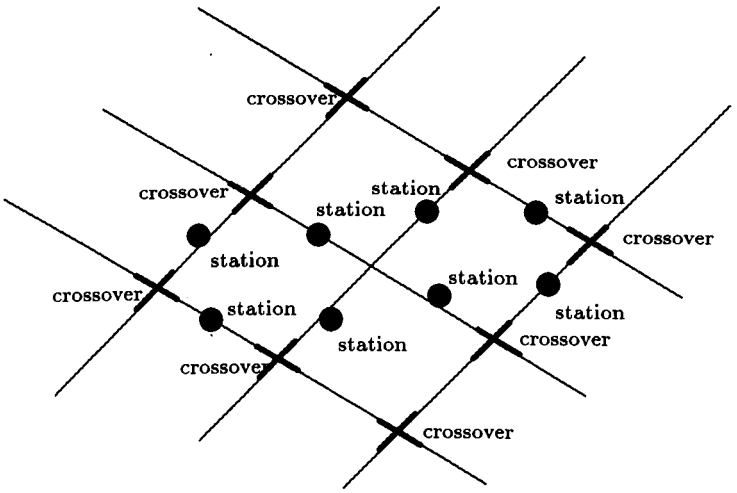


Figure 4.21: Portion of the Green Board

Furthermore, during the process of analysis, we have not identified all the abstractions described in this Section at the same time, that is, the abstractions were identified gradually according to our necessity and better understanding of the problem. For instance, during earlier stages of the analysis we have just worked with the section abstraction, ignoring its different kinds (that is, solid and partitioned section). Only later on, in the final iterations of the analysis, we have incorporated the notion of two different kinds of section.

4.5 Design and Implementation of the Basic Model

During object design, the analysis model is refined and it is provided a detailed basis for implementation. We extend the object models that we have produced during analysis to include associations and attributes that objects require for internal processing. However, most of the design is simple and a direct implementation of the analysis model.

A first prototype of the train set system has been implemented using the C++ programming language[150]. The language was chosen mainly because it is a widespread language which supports the essential principles of object orientation, such as, single and multiple inheritance, generic functions and classes, etc. In addition, the language also supports exception handling, although this feature is still not generally available in the market for UNIX platforms.

Table 4.3 shows some measurements of the C++ classes developed during the implementation of the basic prototype. Application-specific classes are, for instance, Controller and Train, whereas examples of basic classes are List, String and Exception. The implementation of the mimic diagram was developed in an independent project by Leroux[92], and then, later on, integrated to the train set system. The mimic was implemented using Tcl/Tk package, which consists of an independent basis interpreted language, called Tcl, with a powerful XWindow library (Tk) attached to it.

Levels	*Number of Classes*	*Number of Lines (appr.)*
Application-specific classes	11	2300
Basic classes	4	1000
Mimic Diagram	1	2800
Total	16	6100

Table 4.3: Basic Prototype of the Train Set System

Now we exemplify the implementation of the train set system by presenting two interface classes of the system: the controller and the train.

Controller Implementation

The controller is the most important class of the train set system, and its main function is to control trains and switches settings of the board. However, because we have carefully separated the state space of the system and distributed behaviour into independent objects, the implementation of the controller was relatively simple.

In other regards, the controller was broken down into many control objects, and it manages information regarding the railway layout, the mimic diagram and the Märklin interface. In addition, the controller interacts with a number of train controllers (one for each train), instructing them to start their journeys and providing assistance when necessary (e.g., stop trains before running into buffers). There are also a number of section objects (one for each section of the board), which handle reservations, as and when requested by the train controllers. So the controller instance holds a list of all trains, a list of all objects in the layout (mainly stations, connectors and sections), and also, information about what sections a given section leads directly to.

The interface for the class **Controller** is based upon the design decisions that we have already made, and it is partially presented in the following class declaration in C++:

```
class Controller {
 private:
  List<TrainEntry>        trainList; // list of trains
  List<ThingEntry>        staList;   // list of stations
  List<ThingEntry>        conList;   // list of connectors
  List<ThingEntry>        secList;   // list of sections
  List<NextSectionEntry> nexList; // list of the next sections a section

  void    parse();
  Vertex* createVertex(Edge *e);
  Vertex* createStation(Edge *edge, int lb);
  Vertex* createConnector(Edge *edge, int lb, char kindCon);
  void    createSection(Section *sc, int lb);
  void    oppositeSections();
  void  insertControlZone(int sId,int hId,List<ControlZoneEntry>*& lcz);
  void  isSectionFree(int tr, int& b, int sectionId);
  void  getSectionObjId(int sectionId, Section*& sc);
  void  getNextSections(int key, List<NSectionEntry>*& lns);
public:
  Controller ();
```

```
  ~Controller ();
  void insertTrain (..);
  void removeTrain (..);
  void startSystem (..);
  void stopSystem (..);
  void lockNextSections (..);
  void lockSection (..);
  void releaseSection (..);
  void occupySection (..);
  void getSectionId (..);
};
```

Train Implementation

Based upon the design decisions expressed in the event flow diagram in Figure 4.20, we can complete the interface of the class **Train**, which is summarised in Figure 4.22. Each engine in the layout has a corresponding control object in the application software. Each train controller has an **identification** and **speed**. They also have a **backSection** and **frontSection**, corresponding to the links of tracks locked behind and ahead, as a security measure to avoid train crashes. Thus, in the basic model, the control zone of a train is composed by the front section. The attribute **currSection** represents the section currently occupied by the train. The basic movement of a train is to travel between stations, locking a new front section and releasing its old back section.

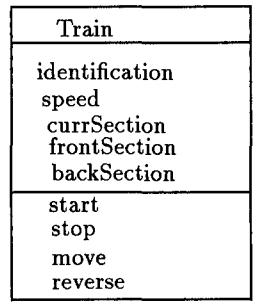


Figure 4.22: Train Class

If, for any reason, a train cannot begin or continue its journey, a default failure exception is signalled to the controller. One possible problem is that a train may be unable to reserve a section because some train is already using it. In this case, the train will stop and wait until the section is free.

4.6 Extending the Basic Model

According to Booch[20], the *evolution* of a system in the object-oriented software development life cycle unifies the traditional aspects of coding, testing and integration. The object-oriented development process results in the incremental production of a series of prototypes, which eventually evolves into the final implementation. Examples of evolutionary changes made during the evolution of a system may include: addition of a new class, change of implementation of a class and reorganization of the class structure. The *maintenance* of a system, on the other hand, involves activities a little bit different from those executed during the evolution. Maintenance concerns the addition of new functionality or modification of some existing behaviour. If the original object-oriented design is well structured, adding new functionality or modifying some existing behaviour should come naturally. So one indication of a well-structured complex system is that it is resilient to changes.

We have come to the end of our design for the basic prototype of the train set system (see Table 4.1). Now in this section, we consider some improvements to the functionality of the system and discuss how our design withstands the changes of requirements. Our main concern is to build a system which supports tolerance of environmental faults. Up to now, we have discussed the normal behaviour phases of objects. From now on, we discuss their abnormal behaviour phases related to environmental faults. The rest of this section is structured as follows. Section 4.6.1 describes the failure analysis for the train set. Section 4.6.2 discusses the mechanisms utilised in the train set system to provide fault tolerance.

Section 4.6.3 deals with the addition of new requirements concerning the tolerance of switch and sensor faults. Following that, we add more new requirements to our design. In Section 4.6.4 we move our design to the Märklin system with real sensors and switches. Section 4.6.5 reviews the implementation of the class Section. Section 4.6.6 considers a fundamental change in the requirements, that is, distributed boards, and finally, Section 4.6.7 shows the final prototype.

4.6.1 Failure Analysis for The Train Set

We have considered so far the normal or desired behaviour phases of objects. Now we discuss the abnormal behaviour phases that includes the notions of environmental faults, failures, and errors. The investigation of abnormal behaviour phases is termed *failure analysis*[137], and it is highly problem-dependent. The train set is an example of a process control application. In this kind of applica-

tion, the main goal is usually to maintain or recover control of a physical process or plant. Abnormal behaviour phases are formalised in state diagrams in the same way as is the normal behaviour phases. In general, abnormal behaviour phases can be very complex, and the models become correspondingly larger and more complicated as more and more abnormal cases are considered. A key point to keep such complexity under control is to have a well-structured and extensible design. In this thesis, we argue that a clear and nice way of achieving this is to use delegation.

A key issue in extending the state diagrams to cover the abnormal behaviour phases is to identify the sources of failures. Two approaches are proposed: one based on examining the external process, and one based on a systematic walk through of the analysis. In the former, some of the aspects to be considered in looking for sources of failures in the external process are:

- equipment failure or malfunction,
- personnel error,
- sensor failures, and
- actuator failures.

In the latter, we can frequently identify sources of failure by examining the "normal behaviour phases" of the dynamic model in a systematic fashion. In the case of the train set, switches and sensors are the unreliable elements of the layout.

4.6.1.1 Fault and Failure Assumptions

The definition of the sensor and switch faults is crucial because it is the basis for the provision of fault tolerance. Although switches and sensors do not exhibit a complex behaviour, it is not straightforward to define their fault types.

A sensor is normal if it outputs correct detection signals, that is, the bit in the decoder corresponding to a given sensor is set to "1" if a train passes on this sensor, and this bit is reset to "0" when it is read in; otherwise the sensor is said to be faulty. We choose to identify three types of sensor faults:

(i) *stuck_at_0* when a sensor always outputs the "0" signal.

(ii) *stuck_at_1* when a sensor always outputs the "1" signal.

(iii) *uncertain* fault which represents any other faulty state of a sensor which has not been previously identified.

In this fault taxonomy, all faults are simply classified as *uncertain*, with the exception of the *stuck_at* faults.

As far as switch faults are concerned, we again choose to identify three types:

(i) *stuck_at_straight* when a switch has a fixed guidance at the straight position.

(ii) *stuck_at_curved* when a switch has a fixed guidance at the curved position.

(iii) *unfixed_guidance* which represents any other faulty state of a switch which has not been previously identified.

The basic assumptions of the failure analysis that we have made for the train set can be summarised as:

- Switch faults: fixed and unfixed guidance faults.
- Sensors faults: *stuck_at_0*, *stuck_at_1* and *uncertain* faults.
- All elements or devices of the train set are reliable, except sensors and switches.
- The status of any sensor or switch is unknown before the trains start running.
- We assume that two successive faulty sensors along a train's route do not occur.

4.6.2 Environmental Fault Tolerance

Generally speaking, fault tolerance is essential for applications that require extremely high levels of availability or reliability. The achievement of fault tolerance is impossible without redundancy. All fault tolerance techniques depend on redundancy being added to the system. Following Lee and Anderson[91], redundancy involves the use of extra elements in the system which are redundant in the sense that they would not be required if the system could be guaranteed to be free from faults.

In the case of the train set, achieving tolerance of environmental faults involves two issues:

- How should we manage the interactions between trains and sections so that the former are unaware that some of the sections may be faulty?
- How can we reconfigure the system properly to sustain this illusion despite the occurrence of further environmental faults?

The first issue is related to error detection and recovery while the second one is related to fault treatment. In some situations, it may be sufficient to deal with errors; however, a more comprehensive approach to fault tolerance will also provide fault treatment in order to eliminate the sources of errors. Fault treatment involves fault diagnosis and system configuration. Fault diagnosis locates the fault and then decides which kind of fault has occurred. After fault diagnosis, we perform the system configuration/repair by switching an appropriate behaviour variant for the faulty object.

4.6.2.1 Error Treatment

The starting point for all dynamic fault tolerance strategies is the detection of an erroneous state. In the case of the train set, a train is capable of detecting an error in its current position based on an exception handling mechanism. As mentioned earlier, each train has a control zone associated to it and each train is responsible for setting its route within the control zone. So a train knows what is the next station to be triggered. When an unexpected sensor is triggered outside its control zone, the train will signal an exception to the controller.

After an error has been detected it is then necessary to remove the errors from the system state by means of error recovery techniques. For the train set application, we have applied forward error recovery; when a train detects that its route has been deviated, it tries to set a new route within its control zone in order to recover from the error. We will return to this point later on in this Section, explaining with more detail the error treatment performed by the train.

4.6.2.2 Fault Treatment

Errors are the symptoms produced by a fault, so although error treatment can remove the immediate danger of a system failure, such as, train crashes, it is still

necessary to treat the fault to prevent it from continuing to damage the system state. Only if the fault is transient, or if the system can cope with recurrent fault manifestations, will error treatment suffice.

Some assessment must be made of the fault's location before a system can deal with a fault. In our application, a train signals an exception to the Controller, which in turn, signals an exception to the train set operator reporting the error. A diagnostic check of connectors and sensors belonging to the train's control zone should be performed to identify the faulty component(s). In fact our diagnostic check of the control zone relies on manual intervention. (An automatic fault diagnosis program for the train set has been implemented some time ago by Zhou[164], but not in an object-oriented fashion; no attempts have been made to incorporate this program into our system.)

The fault location phase results in the identification that one or more components are faulty. Then system repair consists of performing some configuration. For this to be possible the interconnections between should be dynamically switchable. Our reconfiguration strategy consists of encapsulating the abnormal behaviour phases of "faulty" entities as objects, and developing stand-by variants of this abnormal behaviour phases to replace the behaviour implementation of "faulty" components. The abnormal behaviour variants are statically defined, and runtime access to them is implemented through the delegation concept. Thus specific reconfiguration of components can be easily incorporated to a software system for the tolerance of environmental faults.

In the train set system as soon as a moving train detects a deviation of its route, the train signals an exception to the Controller. One possible strategy for implementing the fault location phase is first to isolate the portion of the layout used by the train. (This means that this portion of the layout is temporaly not in use, and the trains continue to move in the restant of the layout.) Then the components within this portion are manually checked to discover which are the faulty one(s). Once the faulty components are identified, the operator initiates the system reconfiguration, and as soon as it is finished, the isolated portion of the layout is back in use.

Our approach has the ability to reconfigure the system on-line, so precautions should be taken to ensure that the switching mechanisms are not activated accidentally. Once the faulty components are identified, the train set operator should start the process of reconfiguration by reporting to the controller the list of faulty components. The controller then performs the adequate reconfiguration. In the following sections we introduce some fundamental changes in the requirements of the system, and show how our design is resilient to them.

4.6.3 Tolerating Environmental Faults

The adoption of a structured approach to the design of systems is widely recognised as being fundamental if the system complexity is to be controlled and minimised. In designing fault tolerant systems, the issue of controlling complexity is critical; an unstructured approach would decrease the system's reliability by introducing more faults than those treated by tolerance, so that the original aim of providing fault tolerance would be defeated.

The idealised fault tolerant component[91] provides an approach for incorporating fault tolerance into a system in a clean and modular manner, minimising the impact on system complexity. Three kinds of exceptional situation are identified:

Interface Exceptions which are signalled when the interface checks determine that an invalid service request has being made to a component.

Local Exceptions which are signalled when a component believes that it has detected an error which is own tolerance provisions can deal with.

Failure Exceptions which are signalled when a component notifies the system that, despite the use of its own fault tolerance capabilities, it has failed on providing the service requested, but the component reports its failure.

The train set system has been built based on this concept of the idealised fault tolerant component. In our approach, a real-world entity may be modelled by several objects, each one representing a variant of its behaviour, i.e., a different behaviour implementation. The set of variants is determined by the state hierarchy associated with the entity's class hierarchy. Each transmutable class implements an idealised fault tolerant component.

4.6.3.1 Tolerance of Switch Faults

The occurrence of an electro-mechanical fault in a switch of the physical layout changes its behaviour phase. As a consequence, its corresponding switch object in the solution domain should also modify its behaviour phase. As soon as the faulty entity is repaired the object is supposed to return to its normal logical state. Transmutable objects can also be applied to model this phenomenon as in the example shown in Chapter 3. State classes allow the description of state-specific properties, and also permit the programmer specify properties common to a group of states using generalisation. Figure 4.23 shows the class *Switch* and

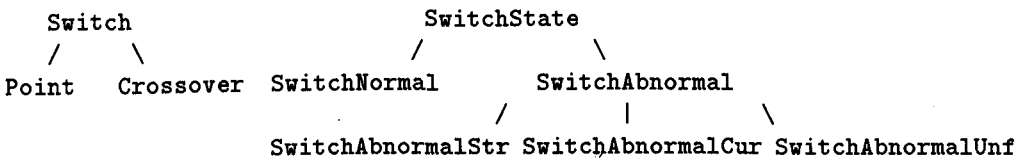


Figure 4.23: Parallel Class Hierarchies for Switch

its corresponding parallel fault hierarchy. A switch can be either in a *normal* or *faulty* state. A faulty switch can suffer three types of faults: *stuck_at_straight*, *stuck_at_curved* and *unfixed_guidance*.

To implement such a change of requirement, only the implementation of the class Switch needs to be altered, but its interface remains unchanged. First, we have to create the new SwitchState hierarchy to model the different behaviour phases of a switch. Then we have to delegate the execution of the method setDirection() to an appropriate variant. Thus, the same abstractions and mechanisms still apply in our design; we have only added the abnormal behaviour phases in an incremental way.

From Analysis to Design. Figure 4.24 shows a partial state diagram for the Connector hierarchy. The class Switch inherits the state diagram of its parent class Connector. A switch can be in SwitchNormal or SwitchAbnormal. To be in SwitchAbnormal, a switch must be in either SwitchAbnormalStr, SwitchAbnormalCur, or SwitchAbnormalUnf.

Consider the object model for the switch hierarchy as shown in Figure 4.25. The refinement of the object model with state classes for the explicit representation of the switch states is shown in Figure 4.25. Refining the object model with state classes would provide a clearer picture of the faulty states of a switch. Method code now very clearly follows the state and the stateclass descriptions. The traditional object-oriented approach would bury the state diagram within the coding of the methods of the Switch class instead of representing such states explicitly in the object model.

From Design to Implementation. A direct implementation of the class Switch described in Figure 4.25 is shown below. The methods are defined in a per-state base, rather than using large case statements at the level of the Switch class. The access to currentState is performed through the operations getMode() and putMode(). The implementation of the operation setDirection() in the class Switch is delegated to the currentState object.

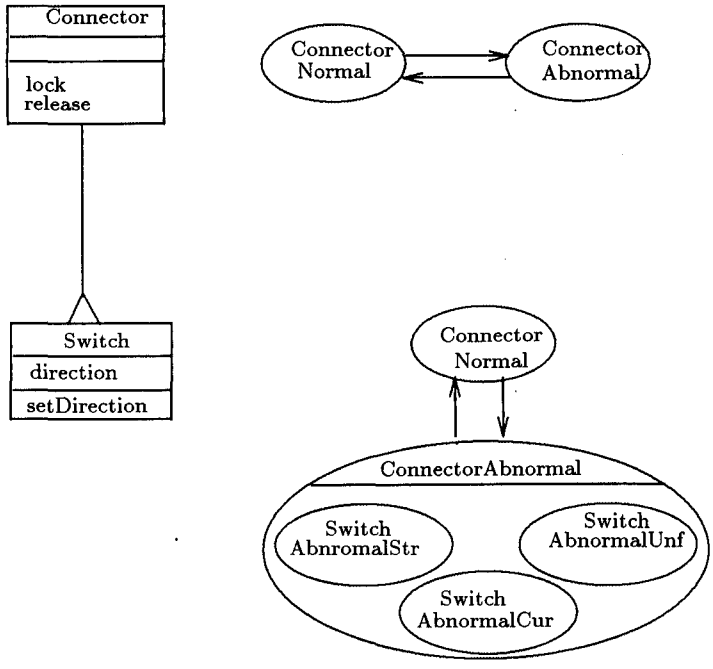


Figure 4.24: State Machine for Connector Hierarchy

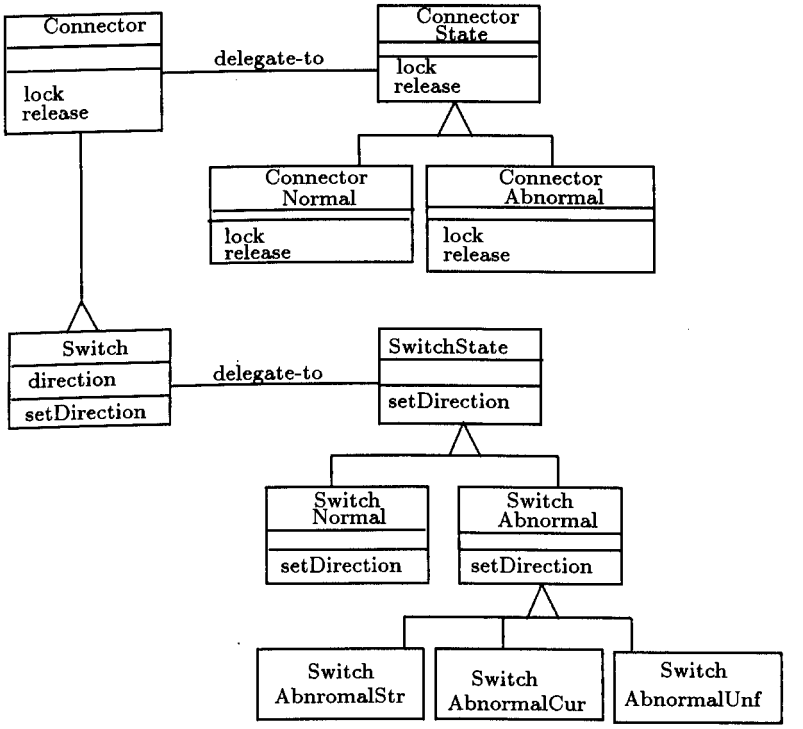


Figure 4.25: State Hierarchies for Connector and Switch

```
enum SwitchSt {SW_NORMAL, SW_ABNORMAL, SW_ABNORMAL, SW_ABNORMAL_CUR,
               SW_ABNORMAL_STR, SW_ABNORMAL_UNF};
enum DirectionType {STRAIGHT, CURVED, NNUULL};

class Switch: public Connector
{
  friend class SwitchState;
  friend class SwitchNormal;
  friend class SwitchAbnormal;
  friend class SwitchAbnormalStr;
  friend class SwitchAbnormalCur;
  friend class SwitchAbnormalUnf;
private:
  SwitchState        *currState;
  SwitchNormal       *normal;
  SwitchAbnormal     *abnormal;
  SwitchAbnormalStr  *str;
  SwitchAbnormalCur  *cur;
  SwitchAbnormalUnf  *unf;
  DirectionType   direction;

  OpHistory* putDirection(DirectionType dir);
  void putMode(SwitchSt m);
public:
   Switch(); // creation of all interface objects
   ~Switch(); // deletion of all interface objects

  OpHistory* setDirection(DirectionType dir){
     OpHistory* oph = new OpHistory;
     *oph += currState->setSwitch(this,dir);
     return oph;
  }

  void getMode(SwitchSt& m){
      currState->getMode(this,m);
  }

  OpHistory* lock(DirectionType dir);
};
```

The implementation of the SwitchState hierarchy is as follows:

```
class SwitchState {
```

```
public:
   SwitchState();
   ~SwitchState();
   virtual OpHistory* setDirection(Switch* sw, DirectionType dir)=0;
   virtual void getMode(Switch* sw, SwitchSt& m)=0;
};

class SwitchNormal: public SwitchState {
public:
   OpHistory* setDirection(Switch *sw, DirType dir){
              sw->putDirection(dir);
              MarklinInterface->setDirection(dir);}

  void getMode(Switch* sw, SwitchSt& m) {
    m = SW_NORMAL;
  }
};

class SwitchAbnormal: public SwitchState {
public:
   virtual void setDirection(Switch *sw, DirType dir){
             signal FAULTY_SWITCH();}

  void getMode(Switch* sw, SwitchSt& m) {
    m = SW_ABNORMAL;
  }
};
```

The substateclass SwitchNormal implements the normal service of the operation setDirection(), while SwitchAbnormal implements the abnormal one. The state-classes derived from SwitchAbnormal which are able to accept an operation must reimplement this function, as shown below:

```
class SwitchAbnormalStr: public SwitchAbnormal {
public:
  virtual OpHistory* setSwitch (Switch* sw, DirType dir){
  OpHistory* oph = new OpHistory;
    if ( dir == STRAIGHT )
    {
      *oph += sw->putDirection(STRAIGHT);
      return oph;
    };
```

```
  *oph += SwitchAbnormal::setSwitch(sw,dir);
  return oph;
  }

  void getMode(Switch* sw, SwitchSt& m) {
    m = SW_ABNORMAL_STR;
  }
};

class SwitchAbnormalCur: public SwitchAbnormal {
public:
  virtual OpHistory* setSwitch(Switch* sw, DirType dir){
  OpHistory* oph = new OpHistory;
  if ( dir == CURVED )
    {
      *oph += sw->putDirection(CURVED);
      return oph;
    };
  *oph += SwitchAbnormal::setSwitch(sw,dir);
  return oph;
  }

  void getMode(Switch* sw, SwitchSt& m) {
    m = SW_ABNORMAL_CUR;
  }
};

class SwitchAbnormalUnf: public SwitchAbnormal {
public:
  void SwitchAbnormalUnf::getMode(Switch* sw, SwitchSt& m) {
    m = SW_ABNORMAL_UNF;
  }
};
```

Extending the Class Train

The discussion so far concerns modifications to perform the treatment of environmental faults. Now we discuss the modifications required to implement the error treatment performed by the train, which includes error detection and error recovery. First, we create a new class, FTConTrain, to model a train which can detect and recover from switch faults (Figure 4.26). The class FTConTrain inherits from Train, and redefines the move() method. The class Train remains unchanged.

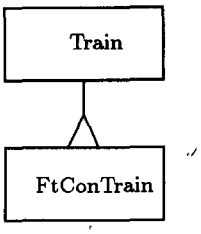


Figure 4.26: Train Hierarchy Extended with Connector Faults

The Train::move() method implements a simple movement of the train around the layout since we assume that all switches and sensors are reliable elements. In this case, it suffices to acquire just one front section to guarantee that a crash would not occur since the layout devices are reliable. However, if we assume that switches may be faulty, a train should acquire all the next sections of its current section in order to maintain its control. If a switch fault occur the train might be deviated from its original path. As a consequence, it is necessary to lock all alternative sections that the train might use. So the notion of control zone plays a fundamental role in the recovery algorithm of the train. The class FTConTrain supports an appropriate set of data structures and operations for the representation of the control zone. Basically, the control zone is a linked list containing references to section objects and holding the necessary information about the next sections.

If we assume that sensors are reliable devices, it suffices to create a two-level control zone to tolerate switch faults. Each train has an assigned route within its control zone. If a train deviates from its original route, that is, an unexpected sensor belonging to its control zone is triggered, the train recovers its current location according to the new location indicated by the triggered sensor. So each train handles its journey by reserving and releasing portions of its control zone as it proceeds. If a train is unable to build its control zone because some other train is already using it, a failure exception will be signalled to the layout controller and the train will stop. Similarly, if a train cannot begin its journey because its control zone is incomplete, a failure exception will be signalled to the layout controller.

4.6.3.2 Tolerance of Switch and Sensor Faults

We assume now that switch and sensor are both unreliable devices of the layout. Our experiment consists of modifying our design which tolerates just switch

faults to create a design which tolerates both switch and sensor faults. The main modification in this case concerns the error treatment algorithm performed by the train. Again, we create a new class, RobustTrain, to model a train which travels around the board detecting and recovering from switch and sensor faults (Figure 4.27). The class RobustTrain inherits from FTConTrain, which in turn inherits from Train. The class FTConTrain remains unchanged.

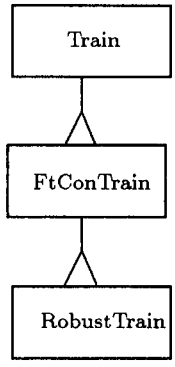


Figure 4.27: Train Module Extended with Connector and Sensor Faults

The class RobustTrain redefines the move method from FTConTrain to implement a control zone with three levels instead of two. If we assume that switch and sensor faults can occur, we need a control zone bigger than when just switches were considered to be unreliable devices. At this point the reader should be reminded we assume that two consecutive faulty sensors along a train's route do not occur. So a robust train needs to build up a control zone with one more level than a switch-fault-tolerant train to ensure that in the case of loosing its way, an error recovery is still possible. If we relax our assumption, for instance, allowing two consecutive sensors to be faulty, the control zone should be enlarge to encompass the new deviations of the train. Consequently, the algorithm for managing the control zone becomes more complex.

In the train set, sensor signals are the only source of information about the trains' locations, so malfunctions of sensors may be a serious obstacles to ensuring that correct system operation is maintained.

4.6.4 Integration of the Low-Level Märklin Interface

The basic prototype of the train set system represents a completed train set simulation, which consists of sixteen domain-specific classes (see Table 4.3). Since

our design already reflects our model of reality, moving to the Märklin system with real sensors and switches involves altering only the implementation of parts of our design. None of the class interfaces of the basic prototype need to be changed. In fact, it is necessary only to modify the implementation of certain classes that lie at the bottom level of the system since we have carefully encapsulated all representation design decisions.

For instance, to modify the class **Switch** to manipulate a physical switch, we need to modify the implementation of the method **setDirection()** so that it sends the appropriate signal to the physical device. We would not have to alter the interface of the class **Switch**. A similar approach applies to the class **SensorPoller**. Our underlying hardware does not interrupt handling, then we have to devise a process that polled the current value of the sensors and then passed the **readSensor()** method. Again, neither the interface nor the semantics of this class change, and therefore no other part of the design need to be altered.

4.6.5 Extending the Class Section

Now let us consider some changes in the requirements of a section. First, suppose that we want to have a controller that is capable of reporting back to a train the availability of a given section as soon as possible. This change is simple to implement. Logically, we just need to alter the implementation of the class **Section**. First, we must create a new state hierarchy, **SectionState**, to model explicitly the states *normal* and *abnormal* of a section (Figure 4.28). Now every time that a switch is detected to be faulty, all sections that have this section in their path should become abnormal.

Since sections are higher abstractions than switches and sensors, if modifications performed in lower-level abstractions are reported to higher abstractions, the efficiency of the system is improved. On the other hand, the implementation of the class **Controller** should also be altered to implement such modification. One solution is create a dependency list of all sections, thus when a switch changes its state, it is possible to know all sections that would require changing.

As we had mentioned earlier, a requirement of the controller is to stop trains running into buffers, i.e., end points. An end point simply means that a section has no next section. Figure 4.29 shows the implementation of the **Section** hierarchy in our basic model. We had created two subclasses, **SolidSection** and **PartitionedSection**, both inheriting from **Section**. Essentially, a *solid* section has next sections, and a *partitioned* section does not have a next section. As a consequence, when the train enters a partitioned section, the controller should stop and reverse it be-

fore hitting the end point. The class PartitionedSection reimplements the method occupy in order to ask to the controller to stop and reverse the train.

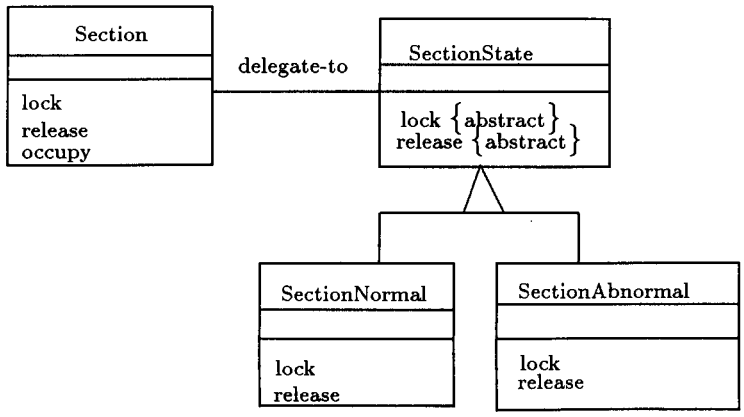


Figure 4.28: State Hierarchy for Section

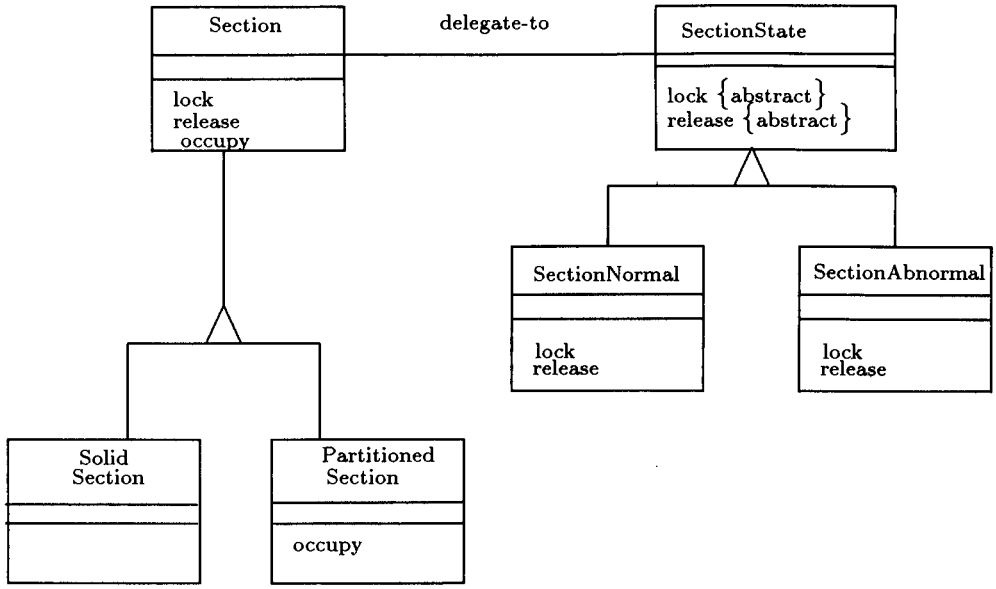


Figure 4.29: Section Hierarchy of the Basic Model

Now let us consider another change in the requirements. Suppose that one wants to have a system which handles interruptions in sections, such as, a tree or a cow in the middle of the section blocking the passage of trains. This change is easy to

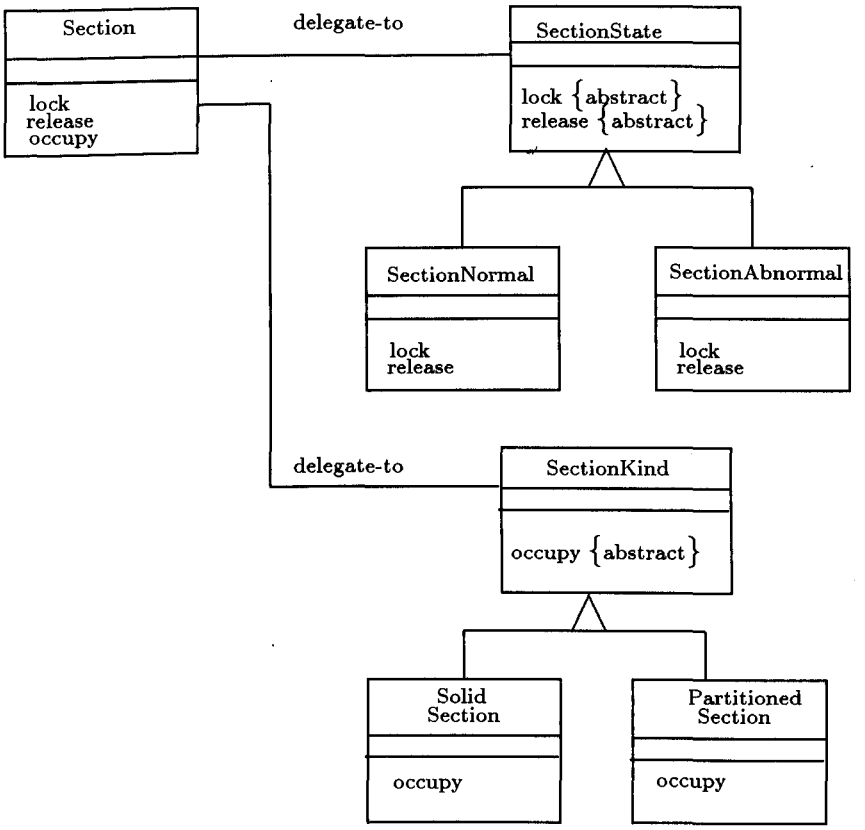


Figure 4.30: Alternative State Hierarchy for Section

implement using the notion of delegation and transmutable objects. The Section hierarchy might be altered as shown in Figure 4.30.

This modification is implemented in a similar fashion as the previous implementations discussed so far, we just moved the classes SolidSection and PartitionedSection to a new hierarchy, SectionKind. The implementation of the method occupy in the class Section is replaced by a new implementation that simply delegates the operation to the *current* kind of section. We moved the original implementation of the method occupy to the class SolidSection, changing every instance variable access to use the original object (which has been passed as an argument to the message). The implementation of the method occupy in the class PartitionedSection is the same, except by the changes in the instance variable access to use the original object.

This simple change makes our design much more flexible. Now if a section is blocked for some reason, the situation can be handled by switching a solid section to a partitioned section. As with all the other changes we have made, no class interfaces need change, which leaves us with a stable design.

4.6.6 Distributed Boards

Now let us consider a fundamental change in the requirements. Suppose that we want to control the three boards individually, each one having its own central unit and its own software controller. So additionally the controllers should manage the layout distribution and also the train crossing between different boards. It should be stressed at this point that each controller operates only on one board and it is unaware of any activity going on in the other boards. However, the controllers can communicate between each other, and the system supports an adequate data structure for the representation of the section connections in the boundaries of the layout.

At first, it seems difficult to conceive how this new requirement would fit in our design without corrupting it. However further examination shows that this change can be performed in an incremental way. First, we define a new kind of section, the *interconnected* section, to model a section which is located at the boundary. Then we extend the SectionKind hierarchy in Figure 4.30 by creating a new class, InterconnectedSection, derived from class SectionKind.

In fact, one can view a section as being refined in two distinct dimensions. This is shown with two separate hierarchies, SectionKind and SectionState. In this particular case, the reconciliation of the dimensions can be a simple and/or combination: each section is either a solid, a partitioned *or* an interconnected, *and* either normal *or* abnormal. However, the addition of InterconnectedSection class creates a dependency with SectionKind class; an interconnected section needs to redefine the methods lock and release defined in the SectionState hierarchy to deal with the distributed boards.

There are at least two ways of restructuring this model. One solution is to factor in one dimension first (for example, normal and abnormal), then the other (for instance, solid, partitioned and interconnected) using inheritance (see Figure 4.31). Another solution is to factor on one dimension first, then the other using delegation instead of inheritance (see Figure 4.32). The former defines just one level of indirection while the latter defines two level of indirection which can be applied in a recursive fashion. One possible limitation of these solutions is that, in general, duplication of declarations and code can exist. In response we would argue

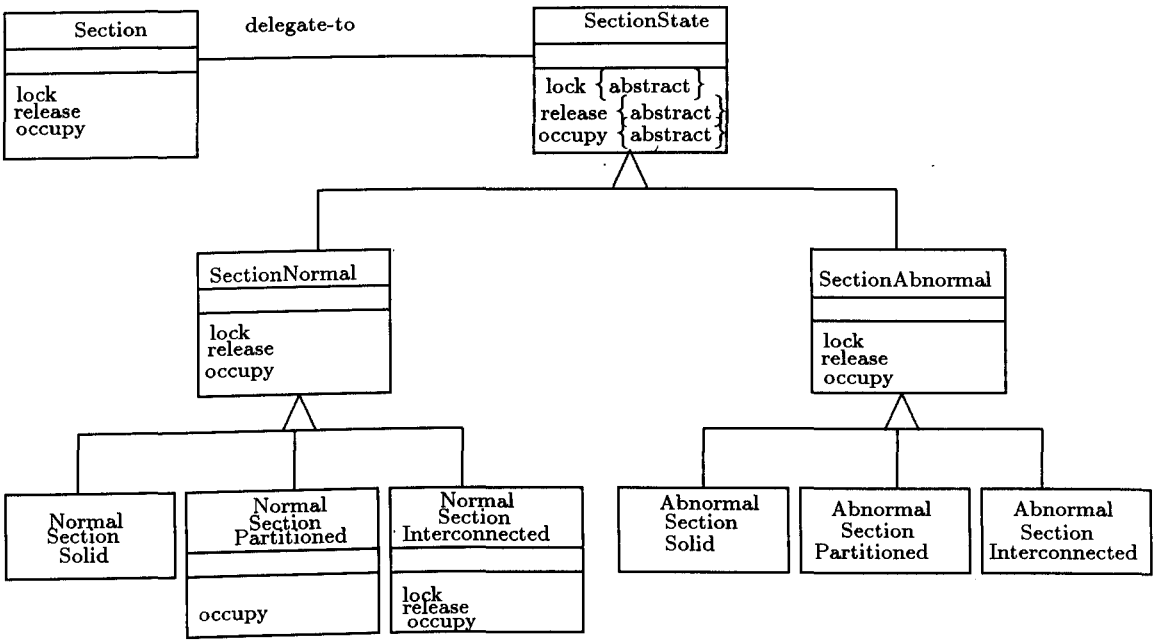


Figure 4.31: State Hierarchy with Distributed Boards Using Inheritance

that our work focuses mainly on the distinction between normal and abnormal behaviour phases, and usually their implementation are rather different. This issue of multiple dimensions or perspectives will be discussed more extensively in the next Chapter.

More specifically, in Figure 4.31 the class **NormalInterconnectedSection** redefines the methods **lock**, **release** and **occupy** to perform correctly these operations in the presence of distributed boards. The method **lock** is performed in two steps: first the lock of the section is performed normally, then the lock of its complementary section located in the neighbour board is requested. If this lock request is attended then the lock of the section is completed with success, otherwise not. The method **release** is implemented in a similar fashion. The method **occupy** is redefined to implement the train crossing between boards. In particular, the implementation of this method would reuse the operations already defined in the class **OperatorInterface**, such as, **insertTrain** and **removeTrain**. In other words, a train crossing can be viewed as its remotion from one board followed by its insertion in the neighbour board.

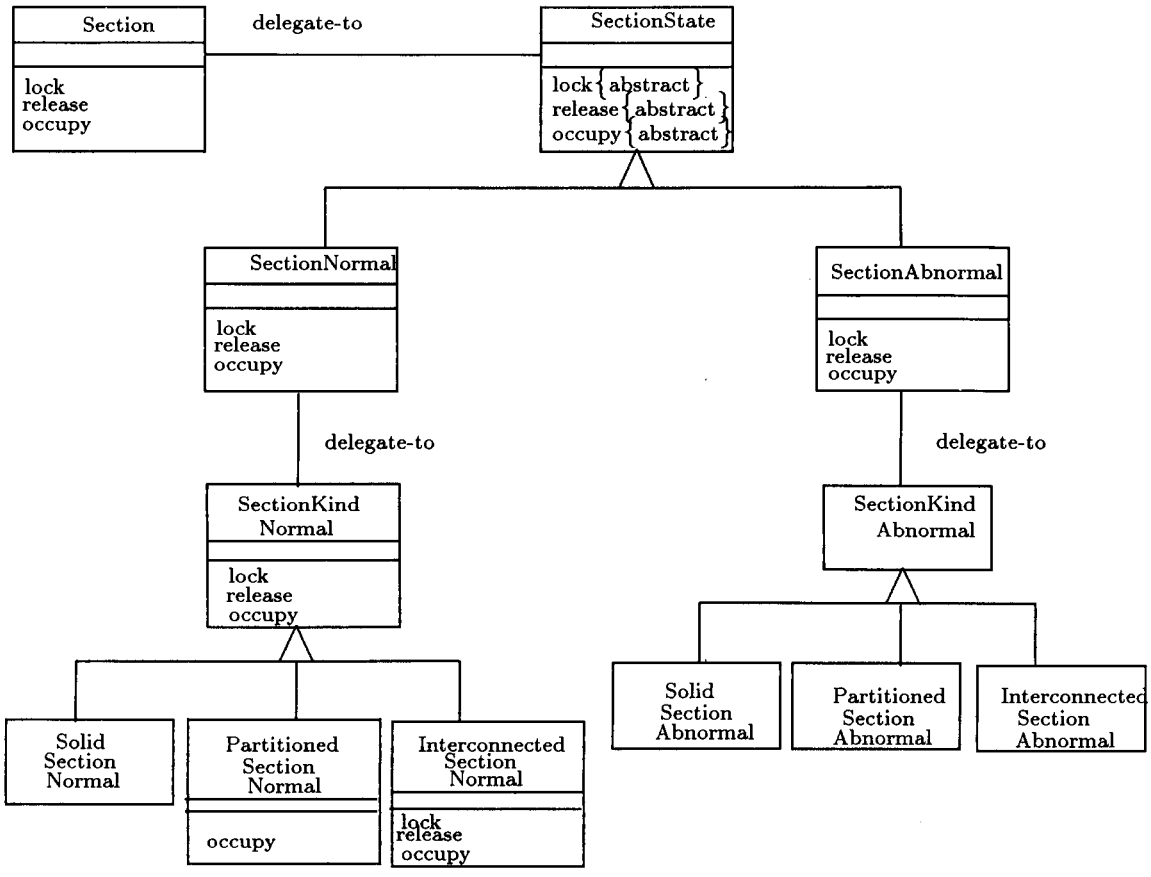


Figure 4.32: State Hierarchy with Distributed Boards Using Delegation

The implementation of the methods **lock**, **release** and **occupy** above described is also applies for the hierarchies of Figure 4.32. As mentioned earlier, the difference between these two solutions is mainly structural; one approach is based in inheritance while the other in delegation, the code of the methods is the same.

4.6.7 Final Prototype

The train set controller fully satisfied its requirements. The final prototype implementation currently incorporates environmental fault tolerance of switches and sensors (see Table 4.4). At the time of writing the integration with the Märklin interface has been partially carried out because the computer interface has some problems in reading the sensors properly. As a consequence, we have chosen to concentrate just on the blue board to carry out the final integration with the low-level Märklin interface, so as to be able to to get a correct reading of the sensors by limiting the numbers of the sensors which should be read.

Levels	*Number of Classes*	*Number of Lines (appr.)*
Basic Prototype	16	6100
Switch and Sensor Faults	15	1650
Low-level Märklin Interface	2	500
Total	33	8250

Table 4.4: Final Prototype of the Train Set System

It can be concluded that the use of delegation as a structuring technique in the second phase of our experiment was essential for "glueing" together class hierarchies and promoting an easy extension of our design.

The implementation of this final prototype can still be improved, especially concerning the implementation of algorithms for building the control zones of trains that have to cope with sensors and switches faults (that is, classes **FTConTrain** and **RobustTrain**). We should find proper algorithms in the literature, possibly in the area of graph theory, for considering all different forms of control zone. The train set layout is very complex with many interconnections which gives rise to many different forms of control zone; in a simpler layout this sort of preoccupation would not exist.

As far as object-oriented methodology is concerning, we were satisfied with our choice of Rumbaugh's OMT methodology for developing our system. This method-

ology proved to be simple and effective with a notation which is easy to understand. I am also particularly fond of Booch's methodology[20], and my impression is that these two methododlogies are going to catch on.

4.7 Experience with the Development of the Controller Prototype

In concluding the implementation of the final prototype of the train set system, we draw the following conclusions from our concrete experience:

- the object-oriented approach can be used very effectively to partition a complex system into manageable pieces.
- the changes and additions were performed in an incremental way without breaking the system, specially due to the use of delegation.
- in general, the use of delegation supports a neat and clear way of extending systems.
- good guidelines to be followed at the design phase are to focus on creating stable interfaces and encapsulate design decisions that are likely to change.
- it is necessary to consider deeply the consequences of the addition of new requirements, and to plan for likely new requirements, knowing that such planning can never be complete.

4.8 Conclusions

> "A complex system that works is invariably found to have evolved from a simple system that worked... A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over, beginning with a working simple system."[54]

We argue that the use of delegation has facilitated not only the provision of environmental fault tolerance but also the addition of new requirements in general. We have demonstrated that by showing how our approach manifested in a real-world application. The experiment have consisted of two main phases:

first, we have developed a basic prototype of the system focusing on the basic requirements, and then extended this model to cope with the new requirements of toleranting environmental faults. We found that adopting such an approach, complexity can be gradually introduced and in a controlled manner.

Many works address the need for excellent interfaces and the benefits of separating interface from implementation so that the implementation can vary. But few investigations consider seriously about the quality of code itself. On the contrary, usually the code is put into in a category of things best not discussed; something to be hidden from view so that it can be changed in privacy. In this Chapter, we have addressed the issue of software structuring and discussed extensively not only interface but also the actual code.

As we have mentioned, an ultimate mark of a well-designed, object-oriented system is that it is resilient to modifications. In this Chapter, we have discussed a number of changes in the requirements and shown how such changes affected our design. In Chapter 6, we will consider further extensions to our design that might be required by future users of the train set structure, and discuss how well we expect our design to be able to cope with them.

BLANK PAGE
IN
ORIGINAL

Chapter 5

Additional Examples

> "Some models capture certain aspects of the system better than others, yet no single model is either *right* or *the best.*"[46]

In the previous chapter, an object-oriented approach for tolerating environmental faults was presented. This approach was used to structure a complex application, the train set. In this chapter, it will be shown that our approach can have a general applicability. This will be demonstrated by giving a number of additional examples which illustrate the usefulness of transmutable objects and state classes.

The examples consist of some "benchmark" problems found in the object-oriented literature, such as, representing multiple attributes and behaviour of people and representing hierarchies of mutable geometric shapes. Each example will be introduced by a brief description, and then an outline of the solution will be presented. At this point, it should be emphasised that the examples shown in this chapter are intended to illustrate the flexibility of the delegation technique, covering points not addressed by the previous examples.

This chapter focuses on issues which are more oriented towards methodological aspects of dynamic and multiple classification (refer to Section 2.1.1 for the definition of classification). The rest of this chapter is organised as follows. The next section examines the concept of multiple classification more closely, after which we consider the concept of dynamic classification. Following that, we discuss the relationship between delegation and aggregation, paying particular attention to the concern of encapsulation. Finally, we describe a design framework to realise the concept of transmutable object in a class-based system.

5.1 Multiple Classification

Multiple classification refers to the ability of an object to be an instance of more than one class. For example, an object called Joe can be instance of both Employee and Father classes. Since the two classes are independent, being an Employee does not automatically imply being a Father - and vice versa.

Multiple classification can be implemented using the multiple inheritance mechanism. In a traditional class-based system, the class associated to an object determines the operations and attributes that it can have. Consequently, it may not be possible for an object to inherit from an arbitrary set of classes. For instance, in the diagram of Fig. 5.1, Employee and Father are classes whose combination defines the subclass EmployeeFather. There is no way to have an object inheriting from the classes Employee and Father unless the class EmployeeFather, whose parents are the above classes, be created. The class EmployeeFather is an example of an *intersection class.*

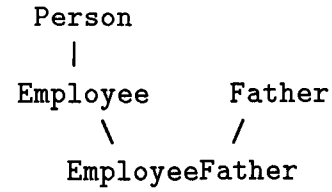


Figure 5.1: Example of an Intersection Class

There are at least two disadvantages with this approach. First, a class is needed for every combination that occur in the system. This can lead to a combinatorial explosion of classes. This problem was a motivation for the creation of "mixins" by the LISP community (see Section 2.2.4.1). Second, not all object-oriented languages support multiple inheritance. Transmutable objects provide a more elegant solution to this problem, as we can see in the example of the following section.

5.1.1 Attributes of Person

A person object might have several independent attributes such as gender and age, and some of the methods on person might depend on the values of these attributes. Figure 5.2 illustrates one possible solution using transmutable objects

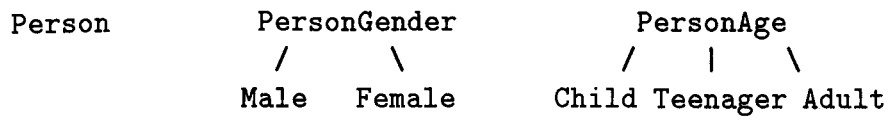


Figure 5.2: State Hierarchiés for Person

and state classes. The C++ implementation for **Person** class might be something like this:

```
class Person {
      friend class PersonGender;
      friend class PersonAge;
private:
  int  sex;
  int  age;
  PersonGender *personGender;
  PersonGender *male;
  PersonGender *female;
  PersonAge    *personAge;
  PersonAge    *child;
  PersonAge    *teenager;
  PersonAge    *adult;
public:
  Person();
  ~Person();

  int bedTime(){ return personAge->bedTime(this);}
  boolean longLived(){ if (age > personGender->expectedLifeTime(this))
                        { TRUE;} else { FALSE;} }
  void haveBirthday(){ age += 1;}
}
```

In this particular example, the bed time of a person varies according to his/her age. A child's bed time could be 8pm, a teenager's bed time might be 10pm, and an adult's bed time can be 11pm. The expected life time of a person varies with his/her gender. The expected life time of a male is 70 while the female is 74.

Transmutable objects provide a clean way of associating behaviour with particular values of object's attributes. There are at least two other different ways of solving the same problem:

1. We could create Male and Female subclasses of the Person class. Then, for example, the attribute *sex* is properly instantiated for each subclass. However, if there are many independent attributes, this solution can lead to a combinatorial explosion of combining subclasses using multiple inheritance, as we have mentioned earlier.

2. Another way is to represent the attributes as instance variables. The methods dependent on the attributes should include conditional constructs, such as if or case statements, to check their values so as to decide which particular branch of the operation is to be performed. This solution would be quite inelegant.

Note that the application of state classes for multiple classification corresponds to the use of aggregation in Harel's statecharts[65], also known as the "and-relationship" (as earlier discussed in Section 3.4.3). Aggregation allows a state to be broken into *orthogonal* components, with limited interaction among them, similar to an object aggregation hierarchy. A state diagram for an aggregate is a collection of state diagrams, one for each component. In the example of Figure 5.2, the state of an object Person consists of AND components, PersonGender and PersonAge. Being a person implies being an aggregation of PersonGender (either Male or Female) and PersonAge (either Child, Teenager or Adult). Concurrency within the state of a single object occurs when the object can be partitioned into subsets of attributes or links, each of which has its own subdiagram. So the state of the object comprises one state of each subdiagram.

5.1.2 Multiple Perspectives

More generally transmutable objects can have multiple perspectives. In our approach, one can link together separate class hierarchies using delegation, with each hierarchy providing a different perspective of the same real-world entity. By such means, it is possible to specify different views of a system's behaviour, and the intended constraints on such behaviour. Traditional class-based inheritance mechanisms are less general in this sense because they force each entity to have a single perspective.

The set of class hierarchies can be regarded simply as co-existing abstractions. One perspective does not have intrinsically more detail or importance than another. So different perspectives may all describe the same aspects of the system from different points of view, at equivalent levels of abstraction. Figure 5.2 shows

an example with two perspectives. The first perspective denotes the entity's sex information whereas the second one denotes the entity's age information.

To summarise, three alternative structuring techniques can be identified along the route of this discussion:

- *Delegation with Aggregate Perspectives.* As shown in Figure 5.2, the best approach using traditional class-based systems is to model **Person** as an aggregate where each component represents a different perspective. This approach is shown in Figure 5.3. Inheritance of operations across the aggregation is implemented using delegation. In this approach, the various intersection classes need not be created explicitly.
- *Inheritance and Delegation.* In Figure 5.4, **Male** and **Female** are subclasses of **PersonGender**. Each subclass is treated as an aggregation of the **PersonAge** hierarchy and the operations are delegated as in the previous approach. This alternative is interesting when one perspective clearly is more important than the others.
- *Intersection Classes.* This approach considers all the possible combinations, as shown in Figure 5.5. If the number of combinations is small, this approach is feasible. However, as pointed out before, this approach has limitations since it promotes duplication of code.

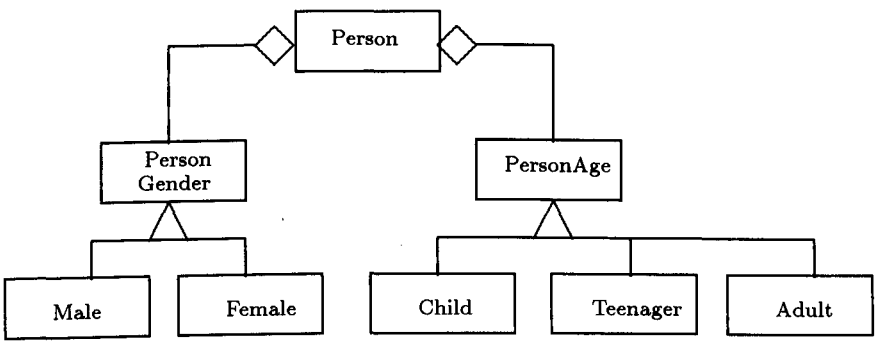


Figure 5.3: Delegation with Aggregate Perspectives

5.2 Dynamic Classification

Dynamic classification refers to the ability to change the class of an object. For

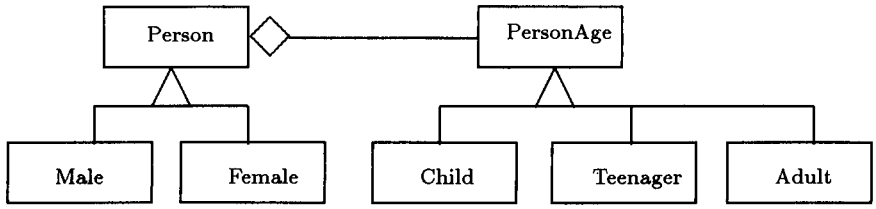


Figure 5.4: Multiple Classification Using Inheritance and Delegation

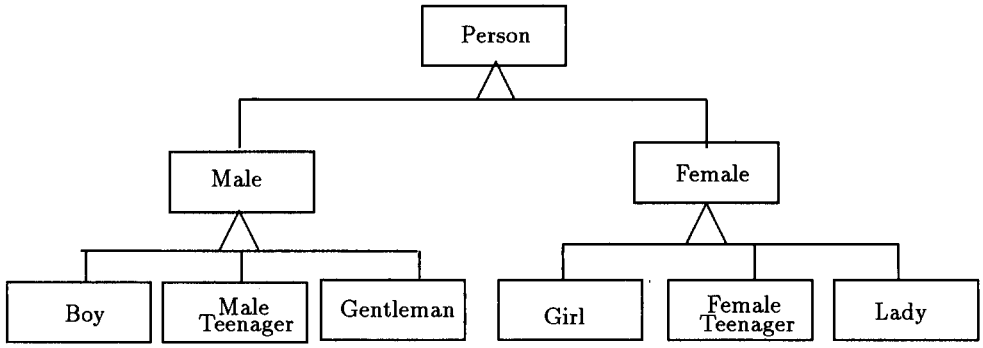


Figure 5.5: Multiple Classification Using Intersection Classes

instance, an object called Joe changes from being an instance of Employee to being a UnemployedPerson. One solution is that whenever an object changes classes a new object is created. Another solution is to define a status flag that indicates the classification. The next two sections discuss two examples where we examine some issues related to dynamic classification.

5.2.1 Buffer

Figure 5.6 illustrates the class hierarchies which implement the class IntBuffer using transmutable objects. (This example is similar to the checking account system presented in Section 3.4.1.) The specification of the class IntBuffer is as follows:

```
class IntBuffer {
      friend class IntBufferState;
private:
  IntBufferState *currentState;
  IntBufferState *empty; // delegatee
  IntBufferState *full; // delegatee
  IntBufferState *partiallyFull; // delegatee
protected:
  int contents[10]; // array of 10 integer elements
public:
  IntBuffer();
  ~IntBuffer();
  virtual int  get(){ return currentState->get(this);}
  virtual void put(int x){ currentState->put(this,x);}
  void updateState(); // state-changing method
}
```

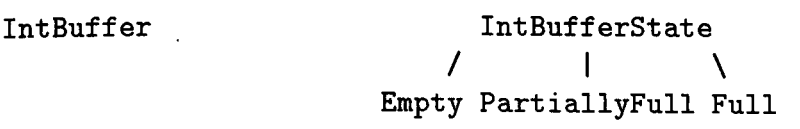


Figure 5.6: Hierarchies for IntBuffer

A IntBuffer object is composed of multiple IntBufferState objects, each one representing a possible perceived state of a buffer: *empty*, *partiallyFull* and *full*. The operations get() and put() are delegated to currentState object for processing. The

object's behaviour is implemented by the delegatees to which the object delegates requested messages.

The IntBufferState hierarchy is implemented as follows:

```
class IntBufferState{
public:
  IntBufferState();
  ~IntBufferState();
  virtual int  get(IntBuffer* b)=0;      // pure virtual
  virtual void put(IntBuffer* b, int x)=0; // pure virtual
}

class Empty: public IntBufferState{
public:
  virtual int  get(IntBuffer* b); // raise error an exception
  virtual void put(IntBuffer* b, int x); // add to back
}

class PartiallyFull: public IntBufferState{
public:
  virtual int  get(IntBuffer* b); // remove from front
  virtual void put(IntBuffer* b, int x); // add to back
}

class Full: public IntBufferState{
public:
  virtual int  get(IntBuffer* b);   // remove from back
  virtual void put(IntBuffer* b, int x); // raise an exception
}
```

Now suppose that we create a subclass called CircularIntBuffer which inherits from IntBuffer (see Figure 5.7). This new class implements a circular buffer with insertPosition and removePosition which cycle around the array *contents*. So the methods get() and put() should be redefined.

The CircularIntBuffer class could be implemented as follows:

```
class CircularIntBuffer: public IntBuffer {
      friend class CircularBufferState;

private:
```

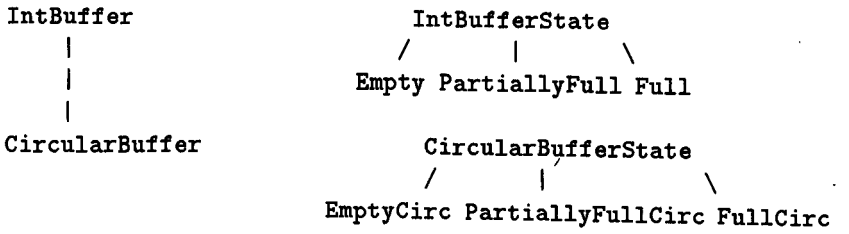


Figure 5.7: Hierarchies for CircularBuffer

```
  int  insertPosition; // an index to the array contents
  int  removePosition; // another integer index

  void store(int elem, int position);
  int retrieve(int position);

  CircularBufferState *currentState;
  CircularBufferState *emptyCirc; // delegatee
  CircularBufferState *fullCirc; // delegatee
  CircularBufferState *partiallyFullCirc; // delegatee

public:
  CircularBuffer();
  ~CircularBuffer();
  virtual int  get(){ return currentState->get(this);}
  virtual void put(int x){ currentState->put(this,x);}
  void updateCircState(); // state-changing method
}
```

The CircularBufferState hierarchy is implemented as follows:

```
class CircularBufferState{
public:
  CircularBufferState();
  ~CircularBufferState();
  virtual int  get(CircularBuffer* cb)=0;       // pure virtual
  virtual void put(CircularBuffer* cb, int x)=0; // pure virtual
}

class EmptyCirc: public CircularBufferState{
public:
  virtual int  get(CircularBuffer* cb); // raise error an exception
```

```
  virtual void put(CircularBuffer* cb, int x); // store element and
                                               // update insertPosition
}

class PartiallyFullCirc: public CircularBufferState{
public:
  virtual int  get(CircularBuffer* cb); // retrieve element and update
                                        // removePosition
  virtual void put(CircularBuffer* cb, int x); //  store element and
                                               // update insertPosition
}

class FullCir: public CircularBufferState{
public:
  virtual int  get(CircularBuffer* cb); // retrieve element and update
                                        // removePosition
  virtual void put(CircularBuffer* cb, int x); // raise an exception
}
```

The implementation of the CircularBufferState hierarchy can be improved in two ways. First, the non-blocking versions of the methods get() and put() can be factorised into the class CircularBufferState. As a consequence, the methods are no longer declared as *pure virtual* in the class CircularBufferState since they have a concrete implementation.

The second improvement concerns the blocking versions of the methods get() and put() which raise an exception. These implementations are exactly the same implementations found in the IntBufferState hierarchy. It would be interesting to find a way of reusing the code already implemented in the IntBufferState hierarchy. Figure 5.8 shows a possible solution. In this diagram the classes EmptyCirc and FullCirc also inherit from Empty and Full, respectively. So one could implement EmptyCirc::get() in the following way:

```
int EmptyCirc::get(CircularBuffer* cb)
{
   Buffer* b = (Buffer*) cb; // explicit conversion of CircularBuffer*
                             // to Buffer*
   Empty::get(cb);
}
```

Note that the *isA* relationship is used to describe when a state class implies another. If one state class explicitly inherits from another, then the child state

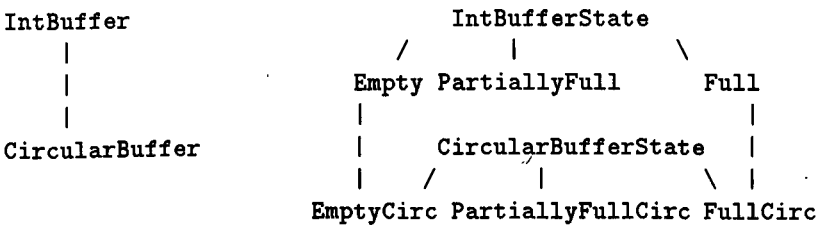


Figure 5.8: Hierarchies for CircularBuffer with Multiple Inheritance

class is assumed to imply the parent state class. Any methods in the child override those in the parent, just like normal classes.

Also note that it is necessary an explicit conversion of object types. The C++ compiler does not keep the information that a CircularBuffer object *isA* IntBuffer. In other words, C++ does not support covariant redefinition of function parameters (discussed in Section 2.2.3). C++ *overloads* the name "get" and allows both definitions, Empty::get(IntBuffer* b) and EmptyCirc::get(CircularBuffer* cb), to exist simultaneously. Even though we read both methods as get(), the compiler treats them as two separate methods. In this way, C++ guarantees that subclasses satisfy the interface of the parent.

To summarise, state classes has increased expressiveness of this example in two ways. First, important states of buffers, such as, *empty* and *full* states, are explicitly identified in the program and named. Moreover, the creation of state classes remind the programmer of the exceptional situations that the code should handle. This is particularly important useful during maintenance and evolution phases of the design when the code is later extended with new functionality.

Second, attaching methods directly to state classes supports better factoring of code and eliminates if and case statements. In the absence of transmutable objects, a method whose behaviour depends on the state of the object would include an if or case statement to identify and branch to the appropriate case; transmutable objects clearly separate the code for each case. By factoring code, separating out the code associated with particular state, we hope to improve the structuring, readability and maintainability of the software.

5.2.2 Transmutable Geometric Shapes

In Figure 5.9, class Polygon is a subclass of Shape, class Rectangle is a subclass of Polygon, and, finally, Square is a subclass of Rectangle. According to mathematical definitions, all squares are rectangles, and all rectangles are polygons, so we obtain the hierarchy described in Figure 5.9. However, if the addVertex is applied to a rectangle object, this object is no longer a rectangle. In a similar fashion, if the method widen is applied to a square object, this object is not anymore a square. So, in this particular example, the multiple values of attributes can trigger a reclassification of shape objects.

```
Shape
  |
Polygon numberOfSides vertices draw() addVertex()
  |
Rectangle draw() widen()
  |
Square draw()
```

Figure 5.9: Standard Hierarchy of Shapes

The Eiffel community has tried to improve things with the notion of *undefine* operations. The solution is to undefine the addVertex operation in the Rectangle class and also to undefine the widen operation in the Square class. However, this approach leads to a complicated, two-phase typing algorithm for guaranteeing static typing checking[106].

This example can be reimplemented using transmutable objects according to Figure 5.10. A polygon object can be either a general polygon, a rectangle or a square. If a vertex is added to a rectangle object, it is reclassified as a general polygon. In a similar way, a square object can be reclassified as a general polygon. If the length of square object is altered, the object becomes a rectangle. In a similar way, if the length of a rectangle object is altered so that its length is equal to its width, this object becomes a square.

5.3 Delegation, Aggregation and Encapsulation

In inheritance, the object structure is built upon the generalisation/specialisation relationship. Specialisation hierarchies typically are treated as type-level constructs. The structure of an inheritance system is static, that is, it is fixed at

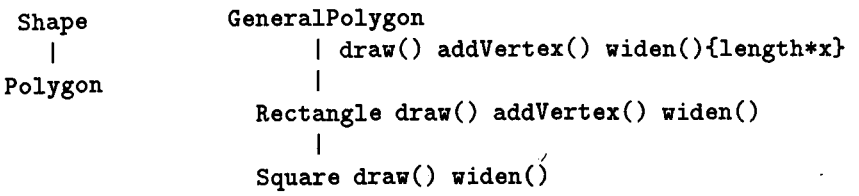


Figure 5.10: Hierarchy of Shapes using Transmutable Objects

the time classes become instantiated. In this thesis, we have also considered specialisation at the level of objects, instead of only classes, using a delegation mechanism. In contrast to inheritance, relationships in delegation are dynamic. In a delegation-based system, primitives are provided for the creation and modification of relationships during runtime.

An important advantage that delegation has over inheritance is that the delegation structure supports *part-whole* links, that is, aggregation. With the exception of some object-oriented knowledge representation languages, such as LOOPS, YAFOOL and OBJLOG[102], and object-oriented database specification languages, such as TROLL[71], object aggregation is a subject neglected by most conventional object-oriented programming languages. However, object aggregation is not only important used for knowledge representation and database, but also for data modelling in general[132, 32]. So it would be useful if major object-oriented languages would provide facilities for it.

We can think of two views of aggregation: a structural and behavioural one. A structural view defines an aggregate object that is structurally composed of its component objects. A behavioural view shows that the behaviour of an aggregate is compose of the behaviour of its components' objects. Property sharing in object composition often has inheritance-like semantics and is occasionally called *horizontal inheritance*, as opposed to inheritance between a class and its subclass which could be called *vertical inheritance*[102].

Figure 5.11 shows the aggregate class **IntBuffer** with the component classes **Empty**, **PartiallyFull** and **Full**, that is, the state classes. The enclosing rectangle stands for the aggregate class which contains in it rectangles standing for the component classes. Note that the aggregate supports the notion of being a recursive module. The external interface of the aggregate is determined by the enclosing rectangle (This diagram is related to Figure 5.6.) The structural properties are expressed by the attributes *empty*, *partiallyFull* and *full*.

Thus, in our approach, we have an aggregate of objects unrelated by inheritance

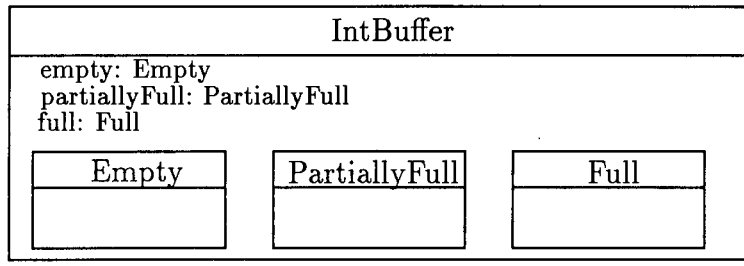


Figure 5.11: Aggregate Class for IntBuffer

(which are in fact related by delegation) which cooperate to perform task. In such a case, the encapsulation mechanism applied is not at the class level since an object in an aggregate may wish to allow others in the aggregate privileged access, whilst maintaining privacy from objects outside the aggregate. Delegation takes place when one object wishes to have another cooperate to complete a task. To accomplish this, the delegatee may need privileged access - just as in real life a subordinate can have a hot line to his superior. The friend construct in C++ and the selective export list in Eiffel are attempts to address this problem placing the access between classes. (Note that in all C++ implementations presented in this thesis we have used the friend construct.)

Note that in Figure 5.11, our "module" is a collection of objects and classes, that is, transmutable objects and state classes. It would be interesting to have language support for modularity and encapsulation when the unit is not limited to a single class. However, no major object-oriented languages, with the exception of Eiffel which has the notion of *cluster*, give support for a building block bigger than a class.

5.4 A Design Framework with Transmutable Classes

Our approach is based on "delegation along the aggregation hierarchy of state classes". Our scheme of delegation is more restrictive: the delegator specifies the executors (i.e., its delegatees) of the delegated task and this information is fixed and checked at compile time. A language with static typing checking, such as C++ and Eiffel, can ensure that a delegatee will understand all the messages delegated to it. What happens is that the delegator chooses one of its delegatees

appropriately to execute a message at run time. But the list of delegatees is already fixed at compile time.

It should be emphasised that our design framework is heavily based on classes and inheritance. First, we use inheritance for the description of the static aspects of the system. Then, delegation is used for the specification of its dynamic aspects related to state-dependent behaviour. A key point is that inheritance of state-dependent behaviour can be decided individually by each transmutable object as with prototype-based systems. The purpose of this thesis was to investigate this combination and to show how it achieves more flexibility without abandoning the advantage of class-based systems.

The following steps were taken in transforming a normal class into a transmutable class in the examples discussed so far (using a class-based language such as C++):

1. identify the observable states.

2. create a state class for each state and organize them in a state class hierarchy. The root class of the hierarchy contains all the delegated methods defined as abstract operations.

3. declare an instance variable in the transmutable class which holds reference to its current state.

4. if a method is state dependent in the transmutable class, move it to the state class hierarchy and substitute it with a new implementation which delegates its execution to the current state. On delegating a message to the current state, add an extra argument in the message which holds a reference to the the delegator. In such a way, the state classes can access the transmutable class.

5. copy all state-dependent methods to every state class, deleting all parts of the code that are for other states and changing every access to use the delegator when appropriate.

After this transformation, one can add a new state by adding a new state class, and the methods of the transmutable class will be spread among the several state classes. Using delegation, an object can change its behaviour since an object's behaviour is implemented bu the objects to which it delegates the request messages. Thus, delegation is useful in building extensible and open systems.

5.5 State of the Art

Some recent investigations have considered the problem of dynamic and multiple classification specially in object-oriented database specification languages[4] and programming languages[29, 151]. Wieringa[157] adopts the terms *class migration* and *role playing* meaning dynamic classification and multiple classification, respectively. He argues that object-oriented models should support three distinct conceptual notions *static subclass*, *dynamic subclass* (class migration) and *role class* (role playing). The basic difference between dynamic subclasses and role classes is that the former have strict identity which is appropriate for inheritance while the latter have their own identifier which is appropriate for delegation. Predicate objects in Cecil[29] also offer support for multiple classification, but there is no explicit distinction between roles and dynamic subclasses. However, the main disadvantage of predicate objects is that they are one more new programming language feature while the main advantage of transmutable objects is that they can be implemented in any class-based object-oriented programming language.

Sciore[135] defines the notion of *roles* in a system mainly based on delegation but also supporting some features of class-based systems. He argues that a more general approach should support the access of an entity with respect to a given role and to allow the entity to dynamically change the roles that it plays. As a consequence, all the objects related to a single real-world entity are arranged into an explicit object hierarchy. That is, an object hierarchy corresponds to a real-world entity, and each object in the hierarchy denotes a role played by the entity. Our approach is heavily based on class-based languages with a restricted support of delegation. Conversely, Sciore's approach is heavily based on delegation systems with a restricted support of classes. Also the work by Otten[122] considers a combination of inheritance and delegation in object-oriented database systems, and according to the author's opinion delegation better suits the requirements of design support.

Chapter 6

Conclusions and Further Research

> "If your problem seems unsolvable, consider that you may have a meta-problem."[54]

This thesis has concentrated on the provision of environmental fault tolerance for software systems exploiting object-oriented techniques. The purpose of this concluding chapter is twofold: discussion of the investigation undertaken highlighting its main contributions, and suggestion of possible directions for future research concerning object-oriented fault tolerance.

6.1 Discussion

What underlies the whole discussion of this thesis is that real-world software systems are enormously complex. We have presented an approach based on delegation for constructing components that are able to tolerate environmental faults. Our proposal is a fundamental way of dealing with the complexity of building up such components in a disciplined and modular form.

To some extent, the presented approach allows dynamic reconfiguration of software components with respect to environmental faults at the level of individual objects. We argue that the adoption of such an approach produces well-structured, object-oriented software systems, and we have demonstrated how our ideas are manifested in a real-world application, a train set controller, which fully

satisfied its requirements. More specifically, our experiment has consisted of two distinct phases: the first phase has coped with the normal aspects of the application whereas the second phase has extended the model generated by the first phase by incorporating tolerance of environmental faults.

As we have discussed in Chapter 2, the object-oriented literature offers many different interpretations and points of view, so it has been difficult to give a precise definition for object-oriented concepts. As a consequence, this research has started by giving a characterisation of an object-oriented based on we felt to be the most coherent, comprehensive definitions and concepts found in the literature.

This thesis has focused on the use of object-orientation from an implementation point of view for the construction of dependable systems. To reiterate, the major contributions which have been achieved by this research are:

- Our approach has proved to be an effective way of structuring a complex control system (i.e., the train set) that have to cope with a variety of environmental faults.
- More generally our approach facilitates evolution and modification of software systems, not only the implementation of environmental fault tolerance.
- We believe that this approach fits well with object-oriented structuring methods for supporting hardware and software fault tolerance, but this yet has to be demonstrated.
- Delegation is a key mechanism for extending systems, and it is worth investigating the support of such delegation in class-based languages.
- We have presented a coherent set of definitions and concepts of object-oriented programming, which has been the basis for the development of this thesis, and we hope that it can also be the basis for projects developed by other researchers in the object-oriented field.

However, it is clear that more experiments are still required (particularly in developing very large software application using object-oriented techniques) before the object-oriented paradigm can claim to be a consolidated topic. These claims can only be fully demonstrated when applied to developing substantial software systems. In particular, this thesis has also aimed to contribute to the consolidation of the object-oriented paradigm through the development of a complex control system.

The Assessment of the Train Set Structure

As we have mentioned earlier, a variety of different projects can be envisaged that would make use of the train set structure described in Chapter 4. However, it remains to be demonstrated what future, unplanned extensions/modifications/applications might be performed based on our structuring of the train set control system, and whether it will prove to be as convenient as our expectations. In spite of this, we believe that most extensions can be implemented in a relative straightforward manner because our proposal consists of a simple structure with wide applicability; however, one should consider *very carefully* how such changes would be incorporated in the proposed design. Possible future changes of the requirements might include:

Air Traffic Control: The use of the train set for simulating air traffic control. In such situation, trains would not be allowed to stop or reverse since they are representing aircrafts, so they should be kept running continuously. A possible solution for the incorporation of such a change in our design is to extend the notion of control zone. Each "aircraft" would have associated to it a control zone containing at least one loop, which would be used by the "aircraft" until its next movement is considered to be safe.

Long Trains: During the design and implementation of the train set, we have assumed that the length of a train is smaller than the smallest section in the board; however, in a real situation a locomotive can pull many waggons, and the train's length can not be ignored. In such a case, the algorithm for locking and releasing sections should be modified to take into account the train's length.

Automatic Fault Diagnosis: Another possible extension that might be implemented is an automatic fault diagnosis program, as described in the work of Bang[164]. In this case, it is likely that the proposed structure would be re-used, and probably some abstractions, such as sections, might be expanded.

6.2 Directions for Future Research

This section presents potential areas of future work related to object-oriented fault tolerance. Some important topics particularly related to the scope of this thesis, which we believe to be worthy investigating include: changes on-the-fly

for non-stopping systems, class evolution, reflection, support for delegation in class-based languages, and formal methods.

On-the-fly changes are required in long-lived, non-stopping systems for supporting, for instance, functionality upgrading and dynamic configuration. Our approach supports to some extent dynamic reconfiguration of the system at the level of instances, with the different "forms" of normal and abnormal behaviour established at compile-time. An appropriate environment containing a dynamic linker might therefore be incorporated to add new forms of behaviour (either normal or abnormal) to the application during runtime.

Another area of future work related to changes on-the-fly which might be examined is evolution of classes. As we pointed out before, our approach provides object evolution as a means of behaviour modification at level of instances. When one has a behaviour modification that occurs in all instances of a class, one should consider the aspects involved in class evolution and not only object evolution. So the following issues need to be clarified:

- how an object decides that it is time to evolve,
- how evolution is performed,
- how its fellow objects incorporate the new behaviour, and
- how to promote evolution of the class that the object is derived from so that future instances of the class are created with the new behaviour.

Furthermore, the problem of transmitting such evolution transmission from an evolved class to its subclasses should also be investigated in class evolution mechanisms. Implicit inheritance between superclass and subclass might be too free and it could be necessary to define a mechanism to transmit selectively properties to the subclasses probably based also on delegation. Of course, it is necessary to bear in mind that all these modifications should be performed in a disciplined manner so as to guarantee the correctness of the overall system.

I think that the reflection concept might play a very important role on the subjects discussed above, that is, changes on-the-fly and class evolution. Research on object orientation and artificial intelligence has led to reification and metalevel reasoning constructs that, although not completely understood, allow the creation of useful systems. Those systems should rely on both adaptive development methods and adaptive software mechanisms to enable the reconfiguration required

to obtain flexibility. In this context, reflection might be the appropriate path to be followed for constructing adaptive and flexible systems.

Another area of future work which might be worth examining is support for delegation in class-based languages. Some research has already appeared in this area, for instance, the work of Wolczko[160] on the MUST language, which is based on Smalltalk, and combines the features of inheritance and delegation. The issues related to encapsulation in the presence of inheritance and delegation are discussed with some detail in [161].

An interesting area for further research is that of providing a more formal treatment for object-oriented programming. Formal methods could provide a solid basis for the semantics of the system as a whole and enforce a coherent use of a language across the design phase. As we have mentioned earlier in Chapter 2, there is a large and growing literature of formal methods and specification in programming, much of which can be applicable to object-oriented programming. For instance, the work of Bar-David[11] uses formal specification, more precisely algebraic specification, applied to object-oriented programming in a very balanced way.

6.3 In Conclusion

We have demonstrated that our approach has many benefits, the two most significant probably being that the resultant program is easier to change in the future and easier to maintain. Other possible advantages are increased reliability, lower developments costs and an increase in the probability of being able to re-use pieces of the program in future programs. Moreover, the use of delegation as a structuring technique was a central idea in our work. Its usage can facilitate the extension of a system in order to incorporate not only tolerance of environmental faults but also other changes in the requirements.

We can also conclude that the combination of class abstraction and delegation is very powerful, and represents a significant potential; however, more experimentation is required before this idea can claim to be a mature topic. Finally, in my opinion, object-oriented design is by no means the final word in design methodology; however, it does represent a fusion of some of the best present ideas about building complex systems.

BLANK PAGE
IN
ORIGINAL

References

[1] O. Agesen, J. Palsberg & M.I. Schwartzbach. Type Inference of Self: Analysis of Objects with Dynamic and Multiple Inheritance. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, Kaiserslautern, Germany, Lecture Notes in Computer Science, **707**: 247-267, Oscar M. Nierstrasz (Ed.), July 1993, Springer-Verlag.

[2] G. Agha. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, Massachusetts, 1986.

[3] A. Albano, R. Bergamini, G. Ghelli & R. Orsini. *An Introduction to the Database Programming Language Fibonacci.* Technical Report, no. FIDE/93/94, FIDE2: Fully Integrated Data Environment, ESPRIT BRA PROJECT 6309, 1993.

[4] A. Albano, R. Bergamini, G. Ghelli & R. Orsini. *An Object Data Model with Roles.* Technical Report, no. FIDE/93/65, FIDE2: Fully Integrated Data Environment, ESPRIT BRA PROJECT 6309, 1993.

[5] C. Alexander. *Notes on the Synthesis of Form.* Harvard University Press, 1964.

[6] P. America. Inheritance and Subtyping in a Parallel Object-Oriented Language. In *Proceedings of the 1st European Conference on Object-Oriented Programming*, ECOOP'87, Paris, France, Lecture Notes in Computer Science, J. Bézivin, J.-M. Hullot, P. Cointe & H. Lieberman (Eds.), **276**: 234-242, June 1987, Springer-Verlag.

[7] P. America. A Parallel Object-Oriented Language with Inheritance and Subtyping. In *Joint Proceedings of the 5th Annual Conference on Object-Oriented Programming: Systems, Languages and Application and the 4th European Conference on Object-Oriented Programming*, OOPSLA/ECOOP'90, Ottawa, Canada, Special Issue of ACM Sigplan Notices, **25**(10): 161-168, N. Meyrowitz (Ed.), October 1990.

[8] T. Anderson. Fault Tolerant Computing. In *Resilient Computing Systems*, T. Anderson (Ed.), Chapter 1, Collins Professional and Technical Books, 1985.

[9] H. Appoyer. *Control of a Train Set Model by the MARS System.* Internal Report, Institut für Technische Informatik, Technical University of Vienna, Austria, September 1994.

[10] A. Avizienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, **SE-11**(12): 1491-1501, December 1985.

[11] T. Bar-David. *Object-Oriented Design for C++.* P T R Prentice-Hall, 1993.

[12] P.G. Basset. Frame-Based Software Engineering. *IEEE Software*, 4(4): 9-16, July 1987.

[13] K. Beck and R. Johnson. Patterns Generate Architecture. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP'94, Bologna, Italy, Lecture Notes in Computer Science, **821**: 139-149, M. Tokoro & R. Pareschi (Eds.), July 1994, Springer-Verlag.

[14] C. Benoit, M. Benoit, L. Henninger & R. Velly. SPOKE: an Object-Oriented Programming Environment. *Journal of Object-Oriented Programming*, **3**(6): 30-38, February 1991.

[15] M. Benveniste & V. Issarny. *Concurrent Programming Notations in the Object-Oriented Language Arche.* Technical Report, no. 1822, IRISA/INRIA-Rennes, France, 1992.

[16] G.S. Blair, J.J. Gallagher & J. Malik. Genericity vs Inheritance vs Delegation vs Conformance vs ... *Journal of Object-Oriented Programming*, **2**(3): 11-17, September/October 1989.

[17] G.S. Blair. *Basic Concepts III (Types, Abstract Data Types and Polymorphism).* In *Object-Oriented Languages, Systems and Applications*, G.S. Blair, J. Gallagher, D. Hutchison & D. Shepherd (Eds.), Chapter 4, Pitman, 1991.

[18] M. Blaha. Aggregation of Parts of Parts of Parts. *Journal of Object-Oriented Programming*, **6**(5): 14-20, September 1993.

[19] D.G. Bobrow, K. Kahn, G. Kiczales, L. Masinter, M. Stefik & F. Zdybel. CommonLoops: Merging LISP and Object-Oriented Programming. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'86, Portland, Oregon, September/October, Special Issue of ACM SIGPLAN Notices, **21**(11): 17-29, N. Meyrowitz (Ed.), November 1986.

[20] G. Booch. *Object-Oriented Design with Applications.* Benjamin/Cummings Publishing Company, Inc., 1991.

[21] R.M. Burstall, D.B. McQueen & D.T. Sannella. *HOPE: An Experimental Applicative Language.* Technical Report CSR-62-80, Department of Computer Science, University of Edinburgh, May 1980.

[22] R.H. Campbell & N. Islam. A Technique for Documenting the Framework of an Object-Oriented System. In *Proceedings of the Second International Workshop on Object Orientation for Operating Systems*, September 1992.

[23] R.H. Campbell, N. Islam & P. Madany. *Choices, Frameworks and Refinement Revisited.* Technical Report, no. UIUCDCS-R-92-1769, Department of Computer Science, University of Illinois at Urbana-Champaign, December 1992.

[24] L.F. Capretz. *Object-Oriented Design Methodologies for Software Systems.* Ph.D. Thesis, Department of Computer Science, University of Newcastle upon Tyne, November 1991.

[25] L. Cardelli. A Semantics of Multiple Inheritance. In *Proceedings of the International Symposium on Semantics of Data Types*, Sophia-Antipolis, France, Lecture Notes in Computer Science, **173**: 51-67, G. Kahn, D.B. MacQueen & G. Plotkin (Eds.), June 1984, Spring-Verlag.

[26] L. Cardelli & P. Wegner. On Understanding Types, Data Abstraction and Polymorphism. *Computing Surveys*, **17**(4): 471-522, December 1985.

[27] L. Cardelli. *Modula-3 Report (revised)*, Systems Research Center, Digital, number 52, November, 1989.

[28] B. Carre & J.-M. Geib. The Point of View Notion for Multiple Inheritance. In *Joint Proceedings of the 5th Annual Conference on Object-Oriented Programming: Systems, Languages and Application, and of the 4th European Conference on Object-Oriented Programming*, OOPSLA/ECOOP'90, Ottawa, Canada, Special Issue of ACM Sigplan Notices, **25**(10): 312-321, N. Meyrowitz (Ed.), October 1990.

[29] C. Chambers. Predicate Classes. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, Kaiserslautern, Germany, Lecture Notes in Computer Science, **707**: 268-296, Oscar M. Nierstrasz (Ed.), July 1993, Springer-Verlag.

[30] D. de Champeaux & P. Faure. A comparative Study of Object-Oriented Analysis Methods. *Journal of Object-Oriented Programming*, **5**(1): 21-33, March/April 1992.

[31] S. Chiba & T. Masuda. Designing an Extensible Distributed Language with a Meta-Level Architecture. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, Kaiserslautern, Germany, Lecture Notes in Computer Science, **707**: 482-501, Oscar M. Nierstrasz (Ed.), July 1993, Springer-Verlag.

[32] F. Civello. Roles for Composite Objects in Object-Oriented Analysis and Design. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'93, Washington, DC, USA, Special Issue of ACM SIGPLAN Notices, **28**(10): 376-393, A. Paepcke (Ed.), October 1993.

[33] D. Coleman, F. Hayes & S. Bear. Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design. *IEEE Transactions on Software Engineering*, **18**(1): 9-18, January 1992.

[34] J.O. Coplien. *Advanced C++: Programming Styles and Idioms.* Addison Wesley, 1992.

[35] K.N.R. Corner, C.D. Elliott, D.C. Halliday, J.M. Kelly, P.M.N. Lam & K.C. Kim. *A Train Control System for the Newcastle University Computing Laboratory Model Railway.* MSc. Dissertation in Computing Software and Systems Design, Computing Laboratory, University of Newcastle upon Tyne, April 1990.

[36] F. Cristian. Exception Handling. *In Dependability of Resilient Computers*, T. Anderson (Ed.), pp. 68-97, BSP Professional Books, Oxford, 1989.

[37] Q. Cui & J. Gannon. Data-Oriented Exception Handling. *IEEE Transactions on Software Engineering*, **18**(5): 393-401, May, 1992.

[38] O.-J. Dahl, B. Myhrhaug & K. Nygaard. *Simula 67 Common Base Language.* Publication no. S-22 (revised), Norwegian Computing Center, Oslo, October 1970.

[39] O.-J. Dahl, E.W. Dijkstra & C.A.R. Hoare. *Structured Programming.* Academic Press, 1972.

[40] S.R. Davis. C++ Objects that Change Their Types. *Journal of Object-Oriented Programming*, **5**(4): 27-32, July/August 1992.

[41] D. Detlefs, M.P. Herlihy & J.M. Wing. Inheritance of Synchronisation and Recovery Properties in Avalon/C++. *IEEE Computer*, **21**(12):57-69, December 1988.

[42] G.N. Dixon & S.K. Shrivastava. Exploiting Type Inheritance Facilities to Implement Recoverability in Object Based Systems. In *Proceedings of the 6th Symposium on Reliability in Distributed Software and Database Systems*, Williamsburg, VA, pp. 107-114, March 1987.

[43] C. Dony. An Object-Oriented Exception Handling System for an Object-Oriented Language. In *Proceedings of the 2nd European Conference on Object-Oriented Programming*, ECOOP'88, Oslo, Norway, Lecture Notes in Computer Science, **322**: 146-161, S. Gjessing and K. Nyggard (Eds.), August 1988, Springer-Verlag.

[44] C. Dony. Exception Handling and Object-Oriented Programming: Towards a Synthesis. In *Joint Proceedings of the 5th Annual Conference on Object-Oriented Programming: Systems, Languages and Application and the 4th European Conference on Object-Oriented Programming*, OOPSLA/ECOOP'90, Ottawa, Canada, Special Issue of ACM Sigplan Notices, **25**(10): 322-330, N. Meyrowitz (Ed.), October 1990.

[45] D. D'Souza. Navigating Those Learning Curves. *Journal of Object-Oriented Programming*, **5**(6): 21-25, October 1992.

[46] D. D'Souza. From Analysis to Design: Chasm, Gully, or Step?. *Journal of Object-Oriented Programming*, **5**(7): 16-19, November/December 1992.

[47] H. Ehrig & B. Mahr. *Fundamentals of Algebraic Specification.* Springer Verlag, 1985.

[48] M. Eriksson. A Correct Example of Multiple Inheritance. *ACM SIGPLAN Notices*, **25**(7): 7-10, July 1990.

[49] J.-C. Fabre, V. Nicomette, T. Pérennou & Z. Wu. Implementing Fault Tolerant Applications Using Reflective Object-Oriented Programming. *PDCS2 Second Year Report*, Predictably Dependable Computing Systems, Newcastle upon Tyne, England, September 1994.

[50] J. Ferber. Computational Reflection in Class-Based Object-Oriented Languages. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'89, New Orleans, Louisiana, Special Issue of ACM SIGPLAN Notices, **24**(10): 317-326, N. Meyrowitz (Ed.), October 1989.

[51] D.G. Firesmith. Frameworks: The Golden Path to Object Nirvana. *Journal of Object-Oriented Programming*, **6**(6): 6-8, October 1993.

[52] B. Foote & R.E. Johnson. Reflective Facilities in Smalltalk-80. In *Proceedings of the 4th Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'89, New Orleans, Louisiana, Special Issue of ACM SIGPLAN Notices, **24**(10): 327-335, N. Meyrowitz (Ed.), October 1989.

[53] P. Freeman. Reusable Software Engineering: Concepts & Research Directions. In *Tutorial on Software Design Techniques*, P. Freeman & A. I. Wasserman (Eds.), Fourth Edition, IEEE, Silver Spring, 1984.

[54] J. Gall. *Systemantics: The Underground Text of Systems Lore: How Systems Really Work and How They Fail.* Second Edition, General Systemantics Press, Ann Arbor, MI, 1986.

[55] J. Gallagher. Basic Concepts II (Variations on a Theme). In *Object-Oriented Languages, Systems and Applications*, G.S. Blair, J. Gallagher, D. Hutchison & D. Shepherd (Eds.), Chapter 3, Pitman, 1991.

[56] E. Gamma, R. Helm, R.E. Johnson & J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, Kaiserslautern, Germany, Lecture Notes in Computer Science, **707**: 406-431, Oscar M. Nierstrasz (Ed.), July 1993, Springer-Verlag.

[57] E. Gamma, R. Helm, R.E. Johnson & J. Vlissides. *Design Patterns: Micro-Architectures for Reusable Object-Oriented.* Addison-Wesley, 1994. (to appear)

[58] D. Gangopadhyay & S. Mitra. ObjChart: Tangible Specification of Reactive Object Behavior. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, Kaiserslautern, Germany, Lecture Notes in Computer Science, **707**: 432-457, Oscar M. Nierstrasz (Ed.), July 1993, Springer-Verlag.

[59] C. Ghezzi & M. Jazayeri. Preview: Evolution of Concepts in Programming Languages. In *Programming Language Concepts*, Chapter 2, pp. 13, John Wiley, 1982.

[60] C. Ghezzi & M. Jazayeri. An Introductory Semantic View of Programming Languages. In *Programming Language Concepts*, Chapter 3, pp. 33-41, John Wiley, 1982.

[61] C. Ghezzi & M. Jazayeri. Control Structures. In *Programming Language Concepts*, Chapter 5, pp. 151-159, John Wiley, 1982.

[62] A. Goldberg & D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[63] P. Grogono & A. Bennet. Polymorphism and Type Checking in Object-Oriented Languages. *ACM SIGPLAN Notices*, **24**(11): 109-115, Month 1990.

[64] D. Halbert & P. O'Brien. Using Types and Inheritance in Object-Oriented Languages. In *Proceedings of the 1st European Conference on Object-Oriented Programming*, ECOOP'87, Paris, France, Lecture Notes in Computer Science, J. Bézivin, J.-M. Hullot, P. Cointe & H. Lieberman (Eds.), **276**: 20-32, June 1987, Springer-Verlag.

[65] D. Harel. Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, **8**: 231-274, North-Holland, 1987.

[66] D. Harel *et al.* On the Formal Semantics of Statecharts. In *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, IEEE Press, NY, USA, pp. 54-64, 1987.

[67] D. Harel *et al.* STATEMATE: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, **SE-16**(4): 403-414, 1990.

[68] D.M. Harland. *Polymorphic Programming Languages: Design and Implementation.* Ellis Horwood, 1984.

[69] W. Harris. Contravariance for the Rest of Us. *Journal of Object-Oriented Programming*, **4**(7): 10-18, November/December 1991.

[70] F. Hart & A. Hillegass. Bending the Rules. *Journal of Object-Oriented Programming*, **6**(8): 68-71, January 1994.

[71] T. Hartmann, R. Jungclaus & G. Saake. Aggregation in a Behavior Oriented Object Model. In *Proceedings of the 6th European Conference on Object-Oriented Programming*, ECOOP'92, Utrecht, Netherlands, Lecture Notes in Computer Science, **615**: 57-77, O. Lehrmann Madsen (Ed.), June/July 1992, Springer-Verlag.

[72] S.I. Hayakawa. *Languages in Thought and Action.* 5th edition, Harcourt Brace Javanovich, 1990.

[73] W. Haythorn. What is Object-Oriented Design?. *Journal of Object-Oriented Programming*, **7**(1): 67-78, March/April, 1994.

[74] R. Helm, I.M. Holland & D. Gangopadhyay. Contracts: Specifying Behavioral Compositions in Object-Oriented Systems. In *Proceedings of the 5th Annual Conference on Object-Oriented Programming: Systems, Languages and Applications and the 4th European Conference on Object-Oriented Programming*, OOPSLA/ECOOP'90, Ottawa, Canada, Special Issue of ACM SIGPLAN Notices, **25**(10): 169-180, N. Meyrowitz (Ed.), October 1990.

[75] I.M. Holland. Specifying Reusable Components Using Contracts. In *Proceedings of the 6th European Conference on Object-Oriented Programming*, ECOOP'92, Utrecht, Netherlands, Lecture Notes in Computer Science, **615**: 287-308, O. Lehrmann Madsen (Ed.), June/July 1992, Springer-Verlag.

[76] E. Horowitz & J.B. Munson. An Expansive View of Reusable Software. *IEEE Transactions on Software Engineering*, **SE-10**(5): 477-487, May 1984.

[77] W.L. Hürsch. Should Superclasses Be Abstract?. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP'94, Bologna, Italy, Lecture Notes in Computer Science, **821**: 12-31, M. Tokoro & R. Pareschi (Eds.), July 1994, Springer-Verlag.

[78] V. Issarny. An Exception Handling Mechanism for Parallel Object-Oriented Programming: Toward Reusable, Robust Distributed Software. *Journal of Object-Oriented Programming*, **6**(6): 29-40, October 1993.

[79] R.E. Johnson & B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, **1**(2): 22-35, June/July 1988.

[80] R.E. Johnson & J.M. Zweig. Delegation in C++. *Journal of Object-Oriented Programming*, **4**(7): 31-34, November/December 1991.

[81] R.E. Johnson. Documenting Frameworks Using Patterns. In *Proceedings of the 7th Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'92, Vancouver, British Columbia, Canada, Special Issue of ACM SIGPLAN Notices, **27**(10): 63-76, A. Paepcke (Ed.), October 1992.

[82] S. Khoshafian & R. Abnous. Inheritance. In *Object Orientation: Concepts, Languages, Databases, User Interfaces*, Chapter 3, pp. 79-142, Wiley, 1990.

[83] G. Kiczales, J. des Rivieris & D.G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, 1991.

[84] A. Koenig & B. Stroustrup. Exception Handling for C++. *Journal of Object-Oriented Programming*, **3**(2): 16-33, July/August 1990.

[85] S. Krakowiak, M. Meysembourg, H. Nguyen Van, M. Riveill, C. Roisin & X. Rousset de Pina. Design and Implementation of an Object-Oriented, Strongly Typed Language for Distributed Applications. *Journal of Object-Oriented Programming*, **3**(3): 11-22, September/October 1990.

[86] J. Kramer, J. Magee & A. Young. Towards Unifying Fault and Change Management. In *Proceedings of the Second IEEE Workshop on Future Trends of Distributed Computer Systems*, Cairo, September 1990.

[87] S. Lacourte. Exceptions in Guide, an Object-Oriented Language for Distributed Applications. In *Proceedings of the 5th European Conference on Object-Oriented Programming*, ECOOP'91, Geneva, Switzerland, Lecture Notes in Computer Science, **512**: 268-287, Pierre America (Ed.), July 1991, Spring-Verlag.

[88] W.R. LaLonde, D.A. Thomas & J.R. Pugh. An Exemplar Based Smalltalk. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'86, Portland,

Oregon, September/October, Special Issue of ACM SIGPLAN Notices, **21**(11): 322-330, N. Meyrowitz (Ed.), November 1986.

[89] W. Lalonde & J. Pugh. Subclassing, Subtyping and Is-a. *Journal of Object-Oriented Programming*, **3**(5): 57-62, January 1991.

[90] D. Lea. Christopher Alexander: an Introduction for Object-Oriented Designers. *ACM SIGSOFT Software Engineering Notes*, **19**(1): 39-46, October 1993.

[91] P.A. Lee & T. Anderson. *Fault Tolerance: Principles and Practice.* Second, Revised Edition, Springer-Verlag, 1990.

[92] C. Leroux. *Interface for the Train Set.* Internal Report, Department of Computing Science, University of Newcastle upon Tyne, April 1994.

[93] H. Lieberman. Using Prototypical Objects to Implement Shared Behaviour in Object-Oriented Systems. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'86, Portland, Oregon, September/October, Special Issue of ACM SIGPLAN Notices, **21**(11): 214-223, N. Meyrowitz (Ed.), November 1986.

[94] S.B. Lippman. *C++ Primer.* Addison-Wesley, 1989.

[95] B.H. Liskov & A. Snyder. Exception Handling in CLU. *IEEE Transactions on Software Engineering*, **SE-5**(6): 546-558, November 1979.

[96] B.H. Liskov & A. Snyder. Data Abstraction and Hierarchy. *Addendum to the Proceedings*, OOPSLA'87, Special Issue of ACM SIGPLAN Notices, **23**(5):17 -34, May 1988.

[97] B. Liskov & J.M. Wing. A New Definition of Subtyping. In *Proceedings of the 7th European Conference on Object-Oriented Programming*, ECOOP'93, Kaiserslautern, Germany, Lecture Notes in Computer Science, **707**: 118-141, Oscar M. Nierstrasz (Ed.), July 1993, Springer-Verlag.

[98] B. Liskov & J.M. Wing. Specifications and Their Use in Defining Types. In *Proceedings of the 8th Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'93, Washington, DC, USA, Special Issue of ACM SIGPLAN Notices, **28**(10): 16-28, A. Paepcke (Ed.), October 1993.

[99] O. Madsen & B. Magnusson. Strong Typing of Object-Oriented Languages Revisited. *ACM SIGPLAN Notices*, **25**(10): 140-150, October 1990.

[100] P. Madany, N. Islam, P. Kougiouris & R.H. Campbell. *Reification and Reflection in C++: An Operating Systems Perspective*, Technical Report, no. UIUCDCs-R-92-1736 , Department of Computer Science, University of Illinois at Urbana-Champaign, March 1992.

[101] P. Maes. Concepts and Experiments in Computational Reflection. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'87, Orlando, Florida, Special Issue of ACM SIGPLAN Notices, **22**(12): 147-155, N. Meyrowitz (Ed.), October 1987.

[102] G. Masini, A. Napoli, D. Colnet, D. Leonard & K. Tombre. *Object-Oriented Languages.* Academic Press, 1991.

[103] J.D. McGregor & T. Korso. Supporting Dimensions of Classification in Object-Oriented Design. *Journal of Object-Oriented Programming*, **5**(9): 25-30, February 1993.

[104] J.D. McGregor & D.M. Dyer. A Note on Inheritance and State Machines. *ACM SIGSOFT Software Engineering Notes*, **18**(4): 61-69, October 1993.

[105] B. Meyer. *Object-Oriented Software Construction.* Prentice-Hall, 1988.

[106] B. Meyer. *Eiffel: The Language*, Prentice Hall, 1991.

[107] B. Meyer. Design by Contract. In *Advances in Object-Oriented Software Engineering*, D. Mandrioli & B. Meyer (Eds.), Chapter 1, pp. 1-50, Prentice Hall, 1992.

[108] B. Meyer. Applying "Design by Contract". *IEEE Computer*, **25**(10): 40-51, October 1992.

[109] J. Micallef. Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, **1**(1): 12-35, April/May, 1988.

[110] R. Milner. *A Theory of Type Polymorphism in Programming.* Internal Report CSR-9-77, Department of Computer Science, University of Edinburgh, Scotland, September 1977.

[111] R. Milner. *A Proposal for Standard ML.* Internal Report CSR-157-83, Department of Computer Science, University of Edinburgh, Scotland, 1983.

[112] D.E. Monarchi & G.I. Puhr. A Research Typology for Object-Oriented Analysis and Design. *Communications of the ACM*, **35**(9): 35-47, September 1992.

[113] D.A. Moon. Object-Oriented with Flavors. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'86, Portland, Oregon, September/October, Special Issue of ACM SIGPLAN Notices, **21**(11): 1-8, N. Meyrowitz (Ed.), November 1986.

[114] G.J. Myers. *Software Reliability.* John Wiley & Sons, 1976.

[115] G.J. Myers. *Composite/Structured Design.* van Nostrand Reinhold Company, 1978.

[116] P. Nauer & B. Randell (Eds.). *Proceedings of the NATO Software Engineering Conference.* 1968.

[117] M.L. Nelson. An Object-Oriented Tower of Babel. *OOPS Messenger*, **2**(3): 3-11, July 1991.

[118] D.A. Norman. *The Design of Everyday Things.* Doubleday/Currency, 1988.

[119] J.J. Odell. Dynamic and Multiple Classification. *Journal of Object-Oriented Programming*, **4**(8): 45-48, January 1992.

[120] J.J. Odell. Managing Object Complexity, part II: Composition. *Journal of Object-Oriented Programming*, **4**(8): 45-48, January 1992.

[121] J.J. Odell. Six Different Kinds of Composition. *Journal of Object-Oriented Programming*, **6**(8): 10-15, January 1994.

[122] D.B.M. Otten & P.J.W. ten Hagen. *On the Role of Delegation and Inheritance in Object-Oriented Database Systems.* Technical Report, CS-R9032, Centre for Mathematics and Computer Science (CWI), Amsterdam, July 1990.

[123] D.L. Parnas. On the Criteria to Be Used in Decomposing Systems into Modules. *Communications of the ACM*, **15**: 1053-1058, 1972.

[124] H.H. Porter. Separating the Subtype Hierarchy from the Inheritance of Implementation. *Journal of Object-Oriented Programming*, **4**(9): 20-29, February 1992.

[125] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering*, **SE-1**(2): 220-232, June 1975.

[126] B. Randell. Approaches to Software Fault Tolerance. In *Conference Proceedings 25th Anniversary LAAS*, pp. 33-42, Toulouse, France, May 1993.

[127] B. Randell & J. Xu. Object-Oriented Software Fault Tolerance: Framework, Reuse and Design Diversity. *PDCS2 First Year Report*, Predictably Dependable Computing Systems, **1**: 165-184, Toulouse, France, September 1993.

[128] C.M.F. Rubira-Calsavara & B. Randell. Object-Oriented Environmental Fault Tolerance. *PDCS2 Second Year Report*, Predictably Dependable Computing Systems, Newcastle upon Tyne, England, September 1994.

[129] C.M.F. Rubira-Calsavara & R.J. Stroud. Forward and Backward Error Recovery in C++. *Journal of Object-Oriented Systems*, **1**(1): 61-85, October 1994.

[130] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy & W. Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, 1991.

[131] H. Sakai. A Method for Contract Design and Delegation in Object Behavior Modeling. *IEICE Transactions on Information & Systems*, **E76-D**(6): 646-655, June 1993.

[132] J.H. Saunders. A Survey of Object-Oriented Programming Languages. *Journal of Object-Oriented Programming*, **1**(6): 5-11, March/April 1989.

[133] C. Schaffert, T. Cooper, B. Bullis, M. Killian & C. Wilpolt. An Introduction to Trellis/Owl. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'86, Portland, Oregon, September/October, Special Issue of ACM SIGPLAN Notices, **21**(11): 9-16, N. Meyrowitz (Ed.), November 1986.

[134] E. Sciore. Object Specialization. *ACM Transactions on Information Systems*, **7**(2): 101-122, April 1989.

[135] R.W. Sebesta. Exception Handling. In *Concepts of Programming Languages*, Chapter 12, pp. 380-399, Benjamin/Cummings, 1989.

[136] S. Shlaer & S.J. Mellor. *Object Lifecycles: Modeling the World in States.* Yourdon Press, 1992.

[137] S.K. Shrivastava, G.N. Dixon & G. D. Parrington. An Overview of the Arjuna Distributed Programming System. *IEEE Software*, **8**(1): 66-73, January 1991.

[138] H.A. Simon. The Architecture of Complexity. In *The Sciences of the Artificial*, Second Edition, MIT Press, 1981.

[139] G.B. Singh. Single Versus Multiple Inheritance in Object-Oriented Programming. *OOPS Messenger*, **5**(1): 34-43, January 1994.

[140] A. Snyder. Encapsulation and Inheritance in Object-Oriented Programming Languages. In *Proceedings of the 1st Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'86, Portland, Oregon, September/October, Special Issue of ACM SIGPLAN Notices, **21**(11): 38-45, N. Meyrowitz (Ed.), November 1986.

[141] M. Stadel. Object-Oriented Programming Techniques to Replace Software Components on the Fly in a Running Program. *ACM SIGPLAN Notices*, **26**(1): 99-108, January 1991.

[142] L.A. Stein. Delegation is Inheritance. *Proceedings of the 2nd Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'87, Orlando, Florida, Special Issue of ACM SIGPLAN Notices, **22**(12): 138-146, N. Meyrowitz (Ed.), October 1987.

[143] L.A. Stein, H. Lieberman & D. Ungar. A Shared View of Sharing: The Orlando Treaty. In *Object-Oriented Concepts, Databases and Applications*, W. Kim & F.H. Lochovsky (Eds.), pp. 31-48, ACM Press/Addison-Wesley Publishing Co., 1989.

[144] C. Strachey. *Fundamentals Concepts in Programming Languages.* Oxford University Press, Oxford, U.K., 1967.

[145] R.E. Strom & S.Yemini. Typestate: A Programming Language Concept for Enhancing Software Reliability. *IEEE Transactions on Software Engineering*, **SE-12**(1): 157-171, January 1986.

[146] R.E. Strom, D.F. Bacon, A.P. Goldberg, A. Lowry, D.M. Yellin & S.A. Yemini. *Hermes: A Language for Distributed Computing.* Prentice Hall, 1991.

[147] R.J. Stroud. Transparency and Reflection in Distributed Systems. *ACM Operating Systems Review*, **22**(2): 99-103, April 1993.

[148] B. Stroustrup. Parametrized Types in C++. *Journal of Object-Oriented Programming*, **1**(5): 5-16, January/February 1989.

[149] B. Stroustrup. *The C++ Programming Language.* Second Edition, Addison-Wesley, 1992.

[150] A. Taivalsaari. Object-Oriented Programming with Modes. *Journal of Object-Oriented Programming*, **6**(3): 25-32, June 1993.

[151] W. Tracz. Software Reuse Myths. *ACM SIGSOFT Software Engineering Notes*, **13**(1): 18-22, January 1988.

[152] D.A. Turner. *Miranda System Manual.* Research Software, Ltd., Canterbury, England, 1987.

[153] D. Ungar & R.B. Smith. Self: The Power of the Simplicity. In *Proceedings of the 2nd Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'87, Orlando, Florida, Special Issue of ACM SIGPLAN Notices, **22**(12): 227-241, N. Meyrowitz (Ed.), October 1987.

[154] I.J. Walker. Requirements of an Object-Oriented Design Method. *Software Engineering Journal*, **7**(2): 102-113, March 1992.

[155] P. Wegner. Concepts and Paradigms of Object-Oriented Programming. *OOPS Messenger*, **1**(1): 7-87, August 1990.

[156] R. Wieringa, W. de Jonge & P. Spruit. Roles and Dynamic Subclasses: a Modal Logic Approach. In *Proceedings of the 8th European Conference on*

Object-Oriented Programming, ECOOP'94, Bologna, Italy, Lecture Notes in Computer Science, **821**: 32-59, M. Tokoro & R. Pareschi (Eds.), July 1994, Springer-Verlag.

[157] N. Wilde & R. Huitt. Maintenance Support for Object-Oriented Programs. *IEEE Transactions on Software Engineering*, **18**(12): 1038-1044, December 1992.

[158] T. Winograd & F. Flores. *Understanding Computers and Cognition: A New Foundation for Design.* Addison-Wesley, 1986.

[159] M. Wolczko. *Introducing MUST - The Mushroom Programming Language.* Technical Report of the Mushroom Project, Department of Computer Science, University of Manchester, 1988.

[160] M. Wolczko. Encapsulation, Delegation and Inheritance in Object-Oriented Languages. *Software Engineering Journal*, **7**(2): 95-101, March 1992.

[161] Y. Yokote. The Apertos Reflective Operating System: The Concept and Its Implementation. In *Proceedings of the 7th Annual Conference on Object-Oriented Programming: Systems, Languages and Application*, OOPSLA'92, Vancouver, British Columbia, Canada, Special Issue of ACM SIGPLAN Notices, **27**(10): 414-434, A. Paepcke (Ed.), October 1992.

[162] J. Xu, B. Randell, C.M.F. Rubira-Calsavara & R.J. Stroud. Towards an Object-Oriented Approach to Software Fault Tolerance. In *Fault-Tolerant Parallel and Distributed Systems*, IEEE Computer Society Press, October 1994.

[163] Z.B. Zhou & B. Randell. *An Automatic Fault Diagnosis Program for a Computer-Controlled Model Railway: AFDP.* Internal Report, Computing Laboratory, University of Newcastle upon Tyne, February 1991.