An Operational Approach to Object-oriented Software Development

Cheng Fun Fong

Doctor of Philosophy

The University of Aston in Birmingham

May 1993

The University of Aston in Birmingham

An Operational Approach to Object-oriented Software Development

Cheng Fun Fong

Doctor of Philosophy

1993

Summary

The traditional waterfall software life cycle model has several weaknesses. One problem is that a working version of a system is unavailable until a late stage in the development; any omissions and mistakes in the specification undetected until that stage can be costly to maintain. The operational approach which emphasises the construction of executable specifications can help to remedy this problem. An operational specification may be exercised to generate the behaviours of the specified system, thereby serving as a prototype to facilitate early validation of the system's functional requirements. Recent ideas have centred on using an existing operational method such as JSD in the specification phase of object-oriented development. An explicit transformation phase following specification is necessary in this approach because differences in abstractions between the two domains need to be bridged.

This research explores an alternative approach of developing an operational specification method specifically for object-oriented development. By incorporating object-oriented concepts in operational specifications, the specifications have the advantage of directly facilitating implementation in an object-oriented language without requiring further significant transformations. In addition, object-oriented concepts can help the developer manage the complexity of the problem domain during specification, whilst providing the user with a specification that closely reflects the real world and so the specification and its execution can be readily understood and validated.

A graphical notation has been developed for the specification method which can capture the dynamic properties of an object-oriented system. A tool has also been implemented comprising an editor to facilitate the input of specifications, and an interpreter which can execute the specifications and graphically animate the behaviours of the specified systems.

Keywords: executable specification, graphical specification animation, software life cycle, functional validation, rapid prototyping.

Acknowledgements

I would like to thank my parents for giving me this opportunity, and my supervisor Bryan Ratcliff for his advice throughout this research. I also wish to acknowledge the support provided by the Committee of Vice-Chancellors and Principals of the Universities of the United Kingdom under the ORS Award scheme.

Contents

Figures

Chapter 1 Introduction

1.1 Background to Research

The discipline of *software engineering* has emerged in response to a set of circumstances known as the 'software crisis'. In the early days of computing, limitations in hardware restricted the application of software to small and well-defined tasks. Most software was developed and maintained by one person or organisation, the process was unmanaged and documentation was often non-existent. Then "the economics of computing began to change drastically as hardware costs plummeted and computer capabilities rose", making it possible and economical to automate more and more applications of increasing complexity (Booch, 1991). The complexity of software is measured in terms of the scale of the problem to be solved as well as the intellectual difficulty involved. Initial experience in building large systems showed that existing ad hoc techniques used for building small system could not be scaled up and were inadequate in helping developers manage the increase in complexity. As a result, software development projects frequently overran their schedules, cost much more than their budgets, were unreliable, unmaintainable and performed poorly (Sommerville, 1992). As the demand for software increased unabated, the cost of hardware continued to fall whilst the cost of software escalated rapidly. Software became the major liability in the high cost and poor quality of computer-based system development. Software development reached a crisis situation. The software crisis was not limited to software that functioned incorrectly, but encompassed problems associated with how to develop software, how to maintain the growing volume of software, and how to keep pace with the growing demand for more software (Pressman, 1992).

Discipline and techniques were needed which allowed the complexity inherent in large software systems to be controlled. Different techniques have evolved which deal with software as an engineered product that requires planning, analysis, design, implementation, testing and maintenance. Pressman (1992) states that "by combining comprehensive methods for all phases in software development; better tools for automating these methods; more powerful building blocks for software implementation; better techniques for software quality assurance; and an overriding philosophy for coordination, control, and management, we can achieve a discipline for software development — a discipline called software engineering". The term 'software engineering' was first used as far back as the late sixties when at an early conference Bauer (1969) offered this definition: "... the establishment and use of sound engineering principles in order to obtain economically software that is reliable, and works efficiently on real machines".

Methods and techniques can be placed in different categories or *paradigms*, based on the conceptual frameworks which define the abstractions and philosophies which they use. Examples include the *procedural* paradigm in which computation is achieved by invoking operations which affect a set of state values, while the *functional* paradigm is based on 'side-effect-free' mathematical equations which map inputs into outputs. The following sections outline two other paradigms which underpin this research.

The Operational Paradigm

The classic life cycle paradigm, often called the waterfall model (Boehm, 1988), prescribes a sequential approach to software development. This model is widely used but suffers from several weaknesses: iterations of activities are permitted only between immediately adjacent phases in the model and uncertainty in requirements in the early stages of development are not accommodated; a working version of a system is not available until a late stage in the development, and any early mistakes undetected until the working program can be costly. The *operational* paradigm (Zave, 1984; Agresti, 1986) has been proposed as an alternative to the waterfall model to remedy some of the weaknesses in the latter. The distinguishing feature of the operational paradigm is its emphasis on constructing executable specifications. An executable specification may be exercised to generate the behaviour of the specified system. An operational specification can therefore serve as a prototype of a system which can facilitate the validation of the system's functional requirements, thereby potentially reducing the cost of development.

When a satisfactory operational specification is obtained, a series of behaviour-preserving transformations may be performed on the specification to facilitate its implementation in a programming language. The transformation process can potentially be automated, allowing the mapping between a specification and its implementation to be carried out mechanically. This transformation approach contrasts directly with the usual 'manual' approach, which relies on the ingenuity of programmers to 'hand generate' code and which can therefore be more prone to error (Agresti, 1986).

Current development techniques which fall into the operational category include JSD (Jackson, 1983), Paisley (Zave, 1984), Gist (Balzer, Goldman & Wile, 1982), and Me-Too (Henderson, 1986). The operational paradigm is described in more detail in a subsequent chapter.

The Object-oriented Paradigm

Many design methods have been proposed to help manage the complexity involved in the design phase of software development. The most influential of the early methods was top-down structured design, which was directly influenced by traditional procedural languages such as FORTRAN and COBOL, in which the basic unit of decomposition is the subprogram. A program written in one of these languages consists of a hierarchy of subprograms, each calling other subprograms to perform its task. The approach taken by top-down structured design is therefore one of algorithmic decomposition, wherein each module denotes a major step in some overall process. However, Stein (1988) observes that the technique does not scale up well for very complex systems. It is also largely inappropriate for object-oriented programming languages (Booch, 1991).

Object-oriented languages depart from procedural languages in that state and behaviour are not treated as independent elements, but are encapsulated in a single entity called an 'object' which represents a unit of data abstraction. Computation in object-oriented programming is achieved by sending messages to objects to carry out their tasks. The concepts of 'class' and 'inheritance' are employed to facilitate code reuse and classification. Object-oriented design views the world as a set of objects, each modelling some real-world entity and exhibiting its own behaviour. A system is decomposed according to the key abstractions in the problem domain. With object-oriented design, designers are no longer constrained by having to map the problem domain into predefined data and control structures present in an implementation language, but can create a "virtually unlimited range of abstract data types and functional abstractions" (Wiener & Sincovec, 1984). Software design becomes decoupled from the representational detail of data in a system and therefore the system is more resilient to changes to the representational detail (Wiener & Sincovec, 1984). Booch (1991) states that object-oriented decomposition yields smaller systems through reuse and "directly addresses the inherent complexity of software by helping us make intelligent decisions regarding the separation of concerns in a large state space". A more detail description of the object-oriented paradigm and an overview of some current object-oriented methods such as HOOD and work by Booch are provided in subsequent chapters.

1.2 Aims of Research

The prototyping capability offered by the operational approach can potentially benefit the software development process in general, including object-oriented development. Recent ideas have centred on using an existing operational method to create specifications which then provide a basis for object-oriented design (Booch, 1991; Masiero & Germano, 1988). Lewis (1991) has looked at the feasibility of this using JSD and describes transformation strategies which may be used to map JSD specifications into Smalltalk-80 implementations. Transformations are always necessary in this approach because differences in concepts and abstractions between the two domains need to be bridged.

An alternative approach would be to consider merging the operational approach with the object-oriented paradigm; real-world entities in the problem domain would be modelled in an operational specification of an object-oriented system in terms of objects and their behaviour. Functions of a system are then defined in terms of interactions amongst the objects. This approach would obviate the need for transformation. Execution of the operational specification would provide a means for validating the functional requirements of the object-oriented system expressed in the specification. If the execution could be animated, behaviour of individual objects and their interactions would be 'brought to life'. Such an animation facility would be a useful tool for an analyst in understanding real-world entities during specification. A more accurate model of the problem domain can be created in the specification with greater understanding of the domain entities. The implementation of a system and its functions built upon an accurate model of the problem domain in the specification is likely to be more robust to future changes in the system's functionality. With animation, the execution of a specification based on entities in the user's environment would be conceptually easier for the user to understand; the user is, therefore, in a better position to help the analyst in validating the behaviour of objects in the specification and the functional requirements expressed.

The aim of this research is to explore the feasibility of developing an operational specification technique which is also object-oriented, by incorporating object-oriented concepts in operational specifications. An important requirement is that the operational specifications should lend themselves to animation of their execution. Analysis of some current object-oriented methods suggests that a new graphical specification notation needs to be developed which can represent executable semantics present in object-oriented systems to enable execution of specifications, and at the same time can represent adequately dynamic behaviour of objects so that execution can be animated. Different aspects of dynamic behaviour to be represented have to be determined, and guidelines for using the notation will have to be provided.

Another objective of the research is to implement a tool which can support the graphical specification process. An important facility of the tool will be the capability to execute a specification and animate the behaviour of objects in the specification during execution. However, the tool will not embody any rules which would automatically generate code for an implementation. This is a separate goal not lying within the bounds of the current research.

1.3 Thesis Structure

This thesis consists of seven chapters. Chapter 2 provides an introduction to the object-oriented paradigm, starting with a historical overview of the development of object-orientation. Characteristics of the key elements in the paradigm — objects and classes — are described, including object communication via message passing and class inheritance. Important concepts that underlie the paradigm like abstraction, encapsulation and polymorphism are also discussed.

Chapter 3 examines the traditional software life cycle model (the waterfall model) used for developing software systems and highlights the inherent problems associated with the model, including its inadequacy in representing object-oriented development. Alternative life cycle models have been proposed specifically to reflect object-oriented development, and these are described next. Characteristics of the new models emphasise the iterative and incremental nature of object-oriented development. The end of the chapter overviews some current object-oriented analysis techniques and design methods described by several authors.

Chapter 4 describes the operational approach to software development which has been proposed to remedy the problems of the conventional life cycle model described in Chapter 3. An overview of JSD and a brief description of three other existing operational methods are provided. The possibility of using an existing operational method to specify object-oriented systems is discussed, and then an analysis of the object-oriented methods described in Chapter 3 is provided to assess the suitability of using the notation in these methods to create operational specifications of object-oriented systems.

Chapter 5 discusses some issues involved in the representation of dynamic behaviour and executable semantics in operational specifications of object-oriented systems. An object-oriented operational specification method is then introduced; the method uses a new graphical notation which can represent the dynamic behaviour and executable semantics identified in Chapter 4. The activities and issues involved in each of the four development steps of the method are discussed. The different diagrams in the notation are also described in each step. The diagrams created using the notation in the development steps collectively form an executable specification of a system. Finally a small example is given to illustrate the specification technique.

In Chapter 6, the implementation of a tool to support the operational object-oriented development method described in Chapter 5 is presented. The tool comprises an editor which can facilitate the input of graphical specifications and an interpreter which can animate the behaviour of systems when the specifications are executed. The user interface of the editor and the internal representation of specifications employed by the editor are described, together with the behaviour of the interpreter and its animation facility.

In conclusion, Chapter 7 reviews and evaluates the work that has been accomplished and discusses some of its implications. Suggestions of some possible directions for the future development of this research are also provided.

Chapter 2 The Object-oriented Paradigm

2.1 A Brief History of Object-orientation

Rentsch (1982) predicts that object-oriented programming will be to the 90s what structured programming was to the 70s. Object-orientation is not a new concept. Its roots can be traced back as early as the 1960s in Norway, when Kristen Nygaard and Ole-Johan Dahl created the discrete simulation language Simula (Dahl & Nygaard, 1966). Simula first introduced the concept of *class*.

In the 1970s the programming language Smalltalk (Goldberg & Robson, 1985) "carried the object oriented paradigm to a smoother model" (Rentsch, 1982), and also popularised the now fashionable term 'object-oriented'. It was first developed as the software component of the Dynabook project at Xerox Palo Alto Research Centre (PARC). Dynabook (Kay, 1969) was an extension of the Flex (FLexible, EXtendible) machine, and was Alan Kay's vision of a truly personal computer that would be usable by diverse users for all kinds of information management need. A flavour of the functional abstraction of LISP (Winston & Horn, 1981) can also be felt in Smalltalk, although Smalltalk and LISP are quite different as languages.

The 80s saw a growth in interest in graphical user interfaces (GUIs) — notably the WIMP (Windows, Icons, Menus and Pointers) interface pioneered by Xerox and later Apple. This has influenced the development of object-oriented languages such as Smalltalk, whose library of classes includes many specifically designed for building such interfaces. The enthusiasm for GUIs has also contributed to the success of object-oriented programming, chiefly because the inherent reusability of object-oriented code helps to overcome the sheer complexity and the concomitant high cost of building these interfaces. User interfaces can be assembled quickly utilising existing library classes (e.g., windows, scroll bars, dialogue boxes and menus, etc.). The Apple Macintosh exemplifies the amount of work involved in building user interfaces; estimated as representing over 200 man-years of development, much of this effort has been expended on interface construction (Graham, 1990).

Exchange of ideas between the object-oriented programming and the artificial intelligence communities started from the mid 70s. This has, on the one hand, led to object-oriented extensions of AI programming languages, such as Flavors (Moon, 1986), Actors (Agha, 1986) and CLOS (Bowbrow, DeMichiel, Gabriel, Keene, Kiczales & Moon, 1988) (all of which are derived from LISP), with AI programming environments such as KEE (Fikes & Kehler, 1985) and ART also having been influenced by object-oriented ideas. Conversely, AI research into knowledge representation using semantic networks (Quillian, 1968) and frames (Minsky, 1975), which first evolved the concept of inheritance, has benefitted object-orientation in the use of the inheritance mechanism.

Although suitable for interface development, early object-oriented programming languages exhibited rather severe performance problems when used in other application areas. The search to increase efficiency has prompted the development of new languages such as Eiffel (Meyer, 1988), and the extension of existing ones — hence the emergence of hybrids like C++ (Stroustrup, 1986) and Object Pascal (Tesler, 1985). It also became apparent that object-oriented programming languages have significant inadequacies in handling typical data management problems such as persistent objects and concurrent access to data; the move to object-oriented database systems is one response to this handicap (Graham, 1990).

As object-oriented programming matured towards the 90s, the focus of interest shifted to the design of object-oriented systems and, more recently, to requirements analysis associated with object-oriented development; a proliferation of object-oriented design methods, followed by object-oriented analysis techniques, started to appear. This progression mirrors the development of the functional decomposition technology which started with structured programming (Dahl, Dijkstra & Hoare, 1972), followed by structured design (Yourdon & Constantine, 1979) and then finally structured analysis (DeMarco 1978).

Software that is being built is getting increasingly larger and more complex; object-oriented methodologies seem to promise a way to help control and manage this complexity. One trend is towards the incorporation of object-oriented features into existing structured techniques together with the CASE tools that support them. The majority of the current object-oriented design methods are only partial life-cycle methods, concentrating on the design and perhaps the implementation of a system. As such, there is a need to couple them with appropriate requirements and system analysis techniques. Use of an object-oriented analysis method preceding object-oriented development would seem to offer the most coherent and consistent approach.

The complexity of today's systems is increasing in more than one dimension; besides more complicated and demanding user functional requirements, increasingly varied data types will have to be handled. Winblad *et al* (1990) envisage a future in which a system can process image, voice and video in addition to text and numbers. Object-orientation facilitates multimedia computing, given its characteristics of encapsulation and polymorphism.

Increasing emphasis on distributed and open systems poses the challenge of finding means of communication for systems across heterogeneous network environments. An object-oriented approach to this problem appears reasonable given its emphasis on encapsulation, and its message passing mechanism, where applications can be forged dynamically as objects across networks communicate via message sending.

2.2 The Object/Message Model

In the procedural paradigm, computation is accomplished by applying operators on operands, or similarly by invoking procedures with data as parameters. Data and procedures are treated as if they were independent elements. This is not an accurate picture because all procedures "make assumptions about the form of the data they manipulate" (Robson, 1981). A procedure can only accept parameters belonging to types it has been defined for, and therefore a procedure and the data it manipulates are closely related. Procedures are viewed as active; they carry out some predetermined computation on the supplied data, which is regarded as passive. Cox (1984) describes this as the "operator/operand model". The *object/message* model is the basis of the object-oriented paradigm. An object-oriented system is populated solely by *objects* — all computational entities (in the most general sense) are objects. In this model, there is no clear demarcation between data and procedures as in the procedural paradigm.

The Collins English Dictionary (1986) defines an object as "a tangible and visible thing ... a focus or target for feelings, thought, etc. ... anything regarded as external to the mind ...". To this general and informal definition, Booch (1990) includes the idea that, in the object-oriented paradigm, an object models some part of reality, and therefore exists in time and space. Of course, beside real-world objects, there are other objects that are the results of the software design process. Smith & Tockey state that "an object represents an individual, identifiable item, unit or entity, either real or abstract, with a well-defined role in the problem domain" (as quoted in

Booch, 1990). An object can be described as an entity that owns a *state*, that exhibits *behaviour*, and possesses an *identity*.

An object's state encompasses its *attributes* and the values of these attributes. An attribute is a characteristic or quality that contributes to the description of an object. For example, a bank customer object would have attributes such as name, account number and bank balance, etc. Attributes have values, and these values denote other distinct objects. Seidewitz & Stark (1986) describe the relationship between an object and its attribute objects as forming a "parent-child hierarchy", dealing with "... the decomposition of larger objects into smaller component objects". Figure 2.1 shows the parent-child hierarchy of a bank customer object described above. Other authors (Booch, 1990; Pun & Winder, 1989) also use the term "containing relationship" to refer to a parent object containing other objects.

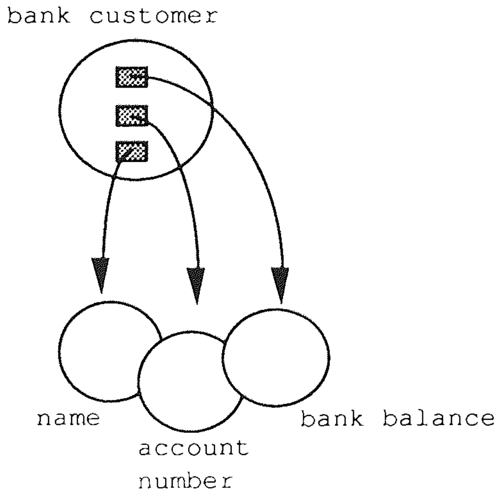


Figure 2.1 Parent-child hierarchy.

All objects are uniform and equal in their status as regard the way they communicate and the means by which they can be referenced; there are no "second class citizens" (Robson, 1981). This means that objects defined and created by programmers have equal status to predefined system objects — *first-class objects* — and can potentially be used by the programmer to implement and manipulate the structuring and processing mechanisms of the programming environment; this represents the potential for a *reflective* system (Maes, 1987).

Object Behaviour

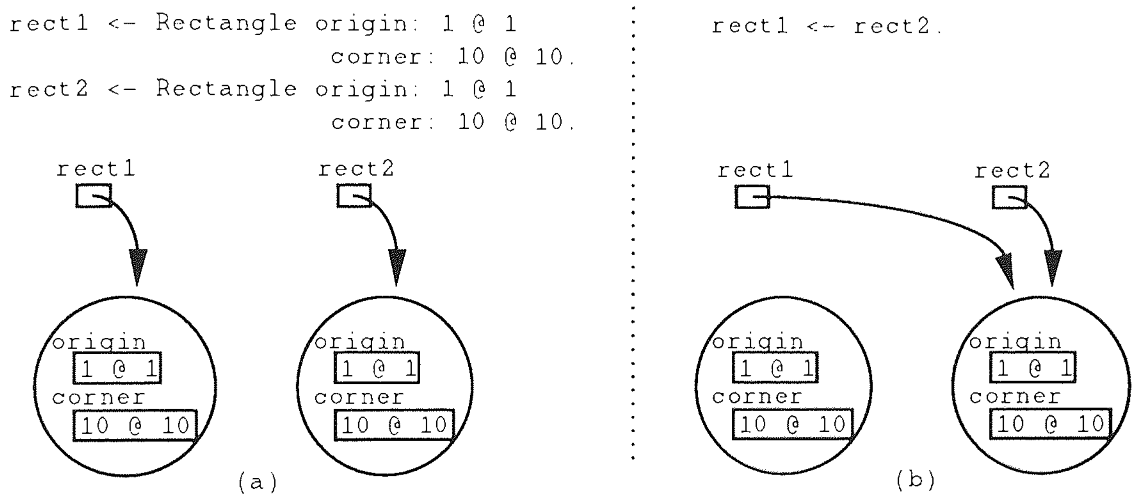
Computation in the object/message model occurs via *message passing* between objects, as opposed to procedure calls and parameter passing. Each object can respond to a set of messages relevant to it; these messages collectively form the object's *protocol.* A message represents a request stating what processing an object (the *receiver*) is requested to carry out, but not how the processing is to be performed. Rentsch (1982) calls this concept "call by desire". This differs from a procedure call which specifies exactly how data is to be operated upon by which piece of code (Ledbetter & Cox, 1985). Some authors (Booch, 1990; Pun &Winder, 1989) state that a "using relationship" exists between two objects when they send each other messages. Seidewitz & Stark (1986) regard the message sending communications of objects as forming a "seniority hierarchy", where objects in a higher layer can use the protocols of objects in all lower layers, but not vice versa; such a hierarchy "deals with the organization of a set of objects into "layers" ". Three possible roles can be played by an object in a using situation (Booch, 1991): *actor* (one which initiates message sending, but is never acted upon by others), *server* (an object that is always a receiver of a message), and *agent* (which can both send and receive messages).

On receiving a message (in its protocol), an object reacts by executing a corresponding *method* of the message; each message is implemented by an associated method. The state of an object can be manipulated only by the object's methods. Thus, access to its private state is restricted to sending the object a valid message. The behaviour of an object is invariably influenced by its state: an object might react differently depending on the values of its attributes. For example, if the bank customer object above receives a debit account message, the request may be rejected if the object's bank balance indicated that the customer had insufficient funds in his account to honour the request. A method might call for the sending of messages to other objects to achieve its task. An object is therefore active and not merely passive data. An object's behaviour is characterised by its protocol and is defined by its methods in terms of its state changes and message communications with other objects. Several types of method can be differentiated (Liskov & Zilles, 1977; Booch, 1990): *modifier*, *selector*, *iterator*, *constructor* and *destructor*. Modifier methods, as the name suggests, modify the state of an object, while selector methods return values of an object's state without interfering with it; an iterator method allows every part of an object's sate to be accessed. Objects are created using constructor methods, and destroyed with destructor methods.

Object Identity

Two objects are similar if they share the same structure and protocol. They are also equal if they contain the same values in their states. The objects are the same, i.e., identical, only if they posses the same *identity*. Every object has a unique identity. Khoshafian & Copeland (1986) explain that identity is "a specific property of an object which distinguishes it from all other objects". The identity of an object must not be based on the values of its states (Tsichritzis & Nierstrasz, 1988). This is an important consideration because the state of an object can change over the object's lifetime, but its identity remains an integral part of the object (Shilling & Sweeney, 1989) and must remain unique (Kersten & Schippers, 1986). However, the use of data values as object identities occurs frequently in database systems where identifier keys are used to identify objects (Khoshafian & Copeland, 1986).

In programming languages, objects can be denoted by variable names. However, there is a danger of confusing addressability with identifiability when names are used to differentiate between objects, and this can often be a source of errors in programming. Problems occur when more than one variable name designates the same object (i.e., the object has one or more aliases); this is know as *structural sharing*. Structural sharing results when the identity of an object is duplicated by assigning one alias of the object to another. The object can then be manipulated via any of its aliases, and this has the side effect, not always intended, of changing the state of the object in relation to all its aliases. For example, consider the Smalltalk code fragment in Figure 2.2(a), where variables `rect1` and `rect2` are assigned two similar and equal `Rectangle` objects, each with a different identity:



```
rect1 <- Rectangle origin: 1 @ 1
                   corner: 10 @ 10.
rect2 <- Rectangle origin: 1 @ 1
                   corner: 10 @ 10.
```

```
rect1 <- rect2.
```

Figure 2.2 Example of object identity.

However, after the execution of the assignment statement (Figure 2.2(b)), structural sharing occurs between `rect1` and `rect2`, and they now refer to the same object with one identity. The object originally designated by `rect1` cannot now be referenced, but still consumes storage space. Some languages like Smalltalk and CLOS automatically destroy objects when they have lost all their references, using the mechanism known as *garbage collection*. Other languages like ADA and C++ put the onus on the programmer to ensure that all redundant objects are destroyed in order to free up storage space.

2.3 Classes

The concept of *class* is tightly associated with the definition of an object. An important characteristic of an object is that each is a unique *instance* of a class. Whereas an object is a concrete entity that exists in time and space, a class is an abstraction — a template from which similar objects may be instantiated. A class describes the attributes of its instances and characterises their behaviour. The attributes of instances are declared as a set of *instance variables* in their class. Each instance of the class has its own individual copy of these variables; instances can therefore own different values that make up their respective states. The interface of a class consists of the messages in the protocol of its instances. *Instance methods* in the class are the implementation of these messages. All instances of a class share the same collection of instance methods; each instance does not keep its own particular versions of the instance methods. Figure 2.3 shows the relationship between a class and its instances.

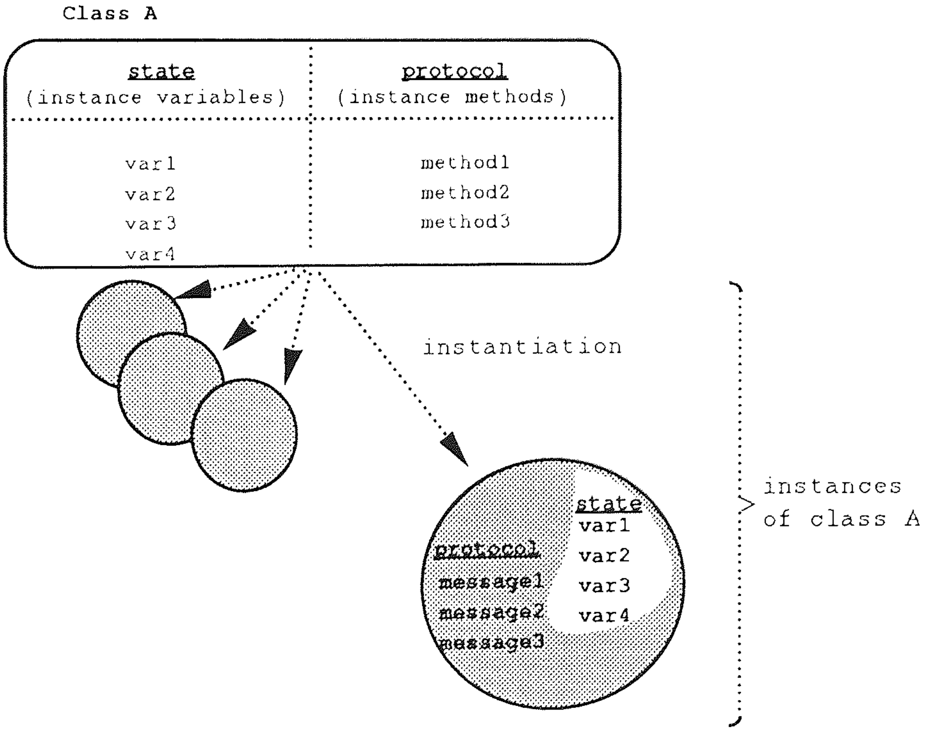


Figure 2.3 A class and its instances.

In some object-oriented languages (e.g., Smalltalk and CLOS), classes are themselves treated as objects, each being regarded as the only instance of its *metaclass*. A class exhibits the same characteristics of an ordinary object: it has class variables, class messages and class methods. However, only a single copy of the class variables is stored for all the instances of a class, so class variables are used for information that is common to all instances of the class, unlike instance variables. Another difference between a class and an instance is that class messages can only be sent to a class itself, not to its instances. The main purpose of a metaclass is to provide the appropriate class method(s) for instantiating instances of the class, and for initialising the class variables. As each class might require a different initialisation for its instances from other classes, a separate metaclass is needed for every class to facilitate this capability. The inheritance phenomenon (to be discussed later) also occurs for metaclasses in terms of variables and methods. Metaclasses are organised into an inheritance structure that parallels that of their classes.

A recent trend has been towards building specialised libraries of classes, called *frameworks* (Deutsch, 1989), to support specific application areas. For example, a user interface framework might incorporate classes such as `Window`, `Scrollbar`, `Button`, etc., which are useful for constructing graphical user interfaces. The first widely used user interface framework was the Model-View-Controller classes of Smalltalk-80 (Krasner & Pope, 1988); MacApp (Schmucker, 1986) is a user interface framework used specifically for implementing Macintosh applications. Most publicised frameworks focus on user interfaces mainly because these frameworks are relatively domain-independent and generally useful to most system developers; other subsystems, apart from user interfaces, tend to be application-specific, hence their frameworks will also need to be specialised.

Wirfs-Brock & Johnson (1990) explain that "a framework is a collection of abstract and concrete classes and the interfaces between them, and is the design for a subsystem". *Abstract classes* are general classes which specify their protocols without fully implementing them (Goldberg & Robson, 1985). Concrete classes are created from abstract classes via the inheritance mechanism, and then adding any required extra attributes, and supplying and/or refining the necessary implementation. "Abstract classes provide a way to express the design of a class" (Wirfs-Brock, 1990). Concrete classes can be instantiated, while abstract classes do not have any actual instances. Winblad *et al* (1990) state that, with the use of appropriate frameworks, the process of building applications can be accelerated and made easier than starting with generic class libraries. Subsystems can be "plugged together" either by using existing concrete classes, or by refining abstract classes. Nevertheless, a framework will not be as generally useful outside its intended application domain.

2.4 Abstraction and Encapsulation

"... the essence of abstraction is to extract essential properties while omitting inessential details" (Ross, Goodenough & Irvine, 1975). Shaw (1980) defines a "good" *abstraction* to be one which emphasises details significant to the user, while suppressing those that are, in the meantime, immaterial. Together with *information hiding* (Parnas, 1972), abstraction provides the greatest leverage for managing complexity in software development (Booch, 1990; Ratcliff, 1987).

Different types of abstraction can be found in different high-level languages. Procedural languages support operational abstractions such as procedures, functions, and operators, and provide computational abstractions to perform iteration and selection. However, Shankar (1984) states: "The nature of abstractions that may be achieved through the use of procedures is well suited to the description of abstract operations, but is not particularly well suited to the description of abstract objects.

This is a serious drawback, for in many applications, the complexity of the data objects to be manipulated contributes substantially to the overall complexity of the problem".

Data types in strongly-typed languages such C and Modula-2 group data values into types based on their representational structures. Variables declared with certain types can only be assigned values of their respective types. This helps facilitate more secure programming (Harland, 1988) through type checking during compilation. Data types alone, however, are inadequate for modeling real world objects because objects "... are apprehended not only through their properties but through their behaviour" (Graham, 1991). Data abstraction characterises data objects not by their structures (types), but by the way they may be manipulated, and therefore provides an instrument for representing external objects; objects in object-oriented languages belong to this type of abstraction. Details of implementation are not essential to the specification of the abstract behaviours of data objects; Cook (1986) states that this represents a clear separation of concerns. For example, consider a set implemented by an array. The abstract point of view is not concerned with how the set is represented; it is interested only in knowing what set operations are available such as union, intersection, membership test etc., and that these operations work. From the implementation point of view, the programmer is concerned with the positions of array elements, and performance issues such as frequencies and patterns of access, and limits on size. *Abstract data types* (Guttag, 1977; Liskov & Zilles, 1975) are specifications of data abstractions which are independent of implementation details. Classes in object-oriented languages which characterise the behaviours of abstract objects are closely related to abstract data types.

As knowledge of implementation is not required in understanding and using a data abstraction, there is no need to include this detail in the specification of the data abstraction. Furthermore, it is advantageous to enforce the concealment of implementation details of data abstractions; this mechanism is know as *encapsulation.*

Abstraction and encapsulation are complementary concepts; abstraction focuses on the outside view of a data abstraction (i.e., the behaviour), while encapsulation prevents clients of the abstraction from seeing its inside view, where the behaviour is implemented. "... encapsulation provides explicit barriers among different abstractions" (Booch, 1990).

Ingalls (1978) states that "no part of a complex system should depend on the internal details of any other part". Encapsulation, also known as information hiding, is an important mechanism for reducing interdependencies between objects (Snyder, 1986). Liskov suggests that "for abstraction to work, implementations must be encapsulated". Modification of internal implementation details is localised and has minimum fallout effects (Pascoe, 1986). Clients rely only on the external interface of classes, and therefore changes to classes can be carried out safely as long as they support the same (or upward compatible) external interface. Encapsulation "allows program changes to be reliably made with limited effort" (Gannon, Hamlet & Mills, 1987). Encapsulation has also made feasible the concept of building reusable software components which can be stored and subsequently retrieved and used when building applications (Meyer, 1987; Cook, 1986, 1987). Ledbetter & Cox (1985) and Cox & Hunt (1986) describe reusable software components as "software-ICs" as an analogy to their hardware counterparts.

2.5 Inheritance

Inheritance is one of the essential elements of object-orientation. Wegner (1987) states that an object-oriented language has to support inheritance. An *object-based* language uses the object as its unit of modularisation; *class-based* languages also implement objects as instances of classes. However, both object-based and class-based languages do not qualify as being object-oriented because they do not support the use of inheritance (Figure 2.4).

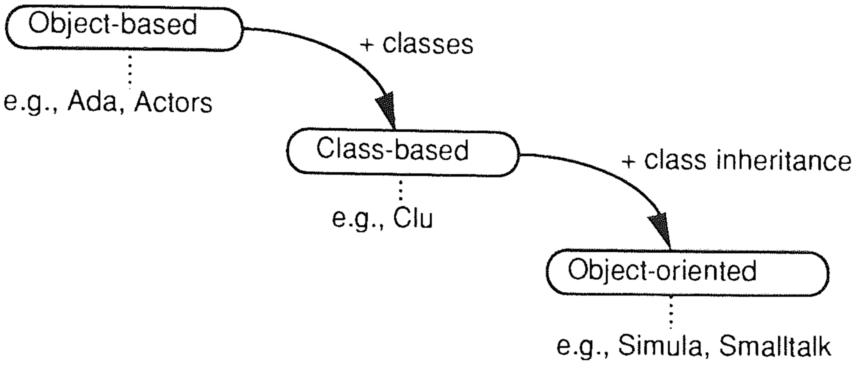


Figure 2.4 Wegner's classification of languages, from object-based to object-oriented.

Inheritance is a mechanism that enables the definition of a new class, or *subclass*, based on an existing class, or *superclass*. In this scheme, classes are organised into an inheritance hierarchy in which classes higher up the structure represent more general abstractions, and classes lower down describe specialisations of the former. Booch (1990) refers to inheritance between classes as a "kind-of" relationship, while some other authors describe it as an "is-a" relationship. To illustrate, consider the example inheritance hierarchy in Figure 2.5:

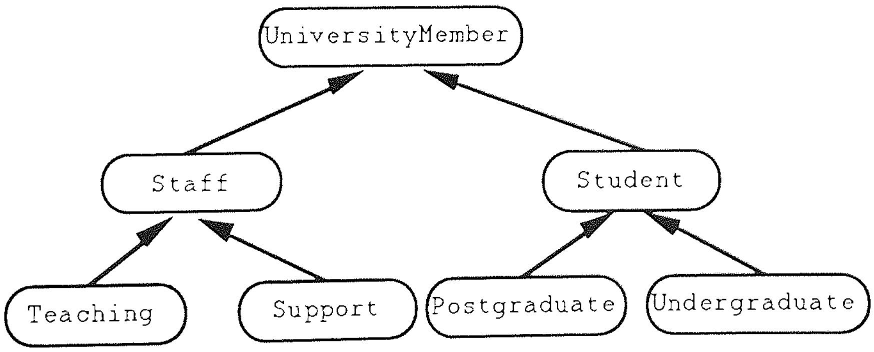


Figure 2.5 An inheritance hierarchy.

Classes `Staff` and `Student` are more specific examples of class `UniversityMember`. They are also classified further: class `Staff` can be categorised into `Teaching` and

Support, and class Student includes the classes Postgraduate and Undergraduate.

A subclass inherits all the properties of its superclass, including the instance variables and methods, and this process propagates upwards until the root of the hierarchy. A subclass specialises by either defining extra instance variables and new methods, or by overriding the behaviours of its superclasses by redefining the implementations of inherited protocols. An object's state therefore consists of the instance variables defined in its own class, plus those in all its superclasses; it can respond to messages in its own class, as well as those of its superclasses. In the example above, class Postgraduate inherits from both class Student and class UniversityMember. Sometimes a few classes might share a subset of their attributes and methods, and yet are not proper subclasses of each other; in this case, a common abstract superclass can be created to subsume their similarities, but the abstract superclass has no meaningful concrete instances of its own. The classes UniversityMember, Staff, and Student would all be abstract classes.

"Inheritance is a mechanism for elision. The power of inheritance is in the economy of expression that results when a class shares description with its superclass" (Stefik & Bobrow, 1986). Inheritance enhances code "factoring" (Pascoe, 1986), leading to a reduction in code bulk. It also eases maintenance (Cox, 1984) because code shared by several classes in an inheritance chain is found in one place only; any change needs only be carried out once, and this change is automatically inherited by all subclasses using the same code. It also facilitates reuse of software and thereby increases productivity since new classes can be derived by inheriting and extending existing classes in an inheritance hierarchy (Pinson & Wiener, 1988).

Used as a classification mechanism, inheritance promotes a better understanding of the relationship between classes in a system. Procedural languages use overloading of operators, but they have difficulty showing the relationships between types. Korson & McGregor (1990) believe that inheritance is the most promising concept available to facilitate constructing software from reusable parts.

Non-strict and strict inheritance (Pun & Winder, 1989a) are two strategies which can be adopted in organising a class inheritance hierarchy.

Non-strict and Strict Inheritance

The aim of non-strict inheritance is to maximise code reuse. A class is chosen as the superclass if it offers the most opportunity for reusing the definition of instance variables and methods. This may have the undesirable effect of introducing some redundant and meaningless variables and methods into the subclass. The subclass may redefine the implementation or block the use of meaningless methods, but nothing can be done about extraneous instance variables. Non-strict inheritance captures only the notion of "similarity" rather than "behavioural" compatibility (Wegner, 1987). For example, a `Stack` and a `Queue` may both include the instance methods `add: item` and `remove` for adding to, and removing an item from, their instances. However, a stack object, by definition, uses a 'last in first out' policy while a queue object applies a 'first in first out' policy; so, although it may be convenient to make one of the classes a subclass of the other, `Stack` and `Queue` are not behaviourally compatible. An inheritance structure constructed using the non-strict inheritance policy can be conceptually confusing and difficult to understand, where semantically unrelated classes can inherit from each other. Wegner (1987) refers to non-strict inheritance between classes as a "like" relationship, and Sakkinen (1989) considers it "incidental" inheritance. Snyder (1986) however, thinks that this form of inheritance has its value, not least because reversing an inheritance decision is facilitated.

Strict inheritance on the other hand, requires descendants to be behaviourally compatible with their ancestors; a subclass must inherit not only the code but the specification of its superclass (Pun & Winder, 1989a), where the specification of a class refers to the signature and the semantics of the methods of the class. The relationship between a superclass and its subclass therefore fulfils the "is-a" relation referred to by Wegner (1987). Sakkinen (1989) states that inheritance of specification is "essential". However, the amount of code reused using strict inheritance could be minimal compared to using non-strict inheritance. This is because new classes may not resemble existing classes close enough (as required by this use of inheritance), to warrant their being made subclasses of the latter; the new classes will need to be added as subclasses of the root class in the inheritance structure, and much of their definition will have to be defined from scratch.

Between the two extremes, Pun & Winder (1989a) suggest a compromise: that an "is-a" relationship should exist between a superclass and its subclass (i.e., they should be behaviourally compatible, and as such classes like `Stack` and `Queue` will not be accepted as subclasses of each other), but the latter is allowed to redefine methods of the inherited protocol where appropriate. This approach is synonymous with conformance in an environment supporting abstract data types. Conformance is closely related to the concept of subtyping; "if any values of type `T` can be used, without requiring a change to their representation, as if they were of type `T'`, then `T` is said to conform to `T'` and `T` is also said to be a subtype of `T'`" (Blair *et al*, 1989). An abstract type P conforms to anther abstract type Q if and only if (Blair *et al*, 1989):

i. P provides at least the operations of Q (P may have more operations)

ii. for each operation in Q, the corresponding operation in P has the same number of arguments and results

iii. the abstract types of the results of P's operations conforms to the abstract types of the results of Q's operations

iv. the abstract types of the arguments of Q's operations must conform to the abstract types of the arguments of P's operations (i.e., arguments must conform in the opposite direction).

Black *et al* (1987) observe that "conformity is a relationship between interfaces"; an abstract subtype can, therefore, have different implementations for its operations from its abstract supertypes.

Multiple Inheritance

In the above description, each subclass has exactly one immediate superclass. This form of inheritance is *single inheritance. Multiple inheritance* removes this restriction: each class can be a subclass of more than one immediate superclass. For example, research students in a university may also be part-time teaching staff, therefore in the previous example in Figure 2.5, a new class `Research` inherits from both `Postgraduate` and `Teaching` (Figure 2.6). The inheritance structure is thus arranged like a lattice (Stefik & Bobrow, 1986) rather than a straight-forward hierarchy.

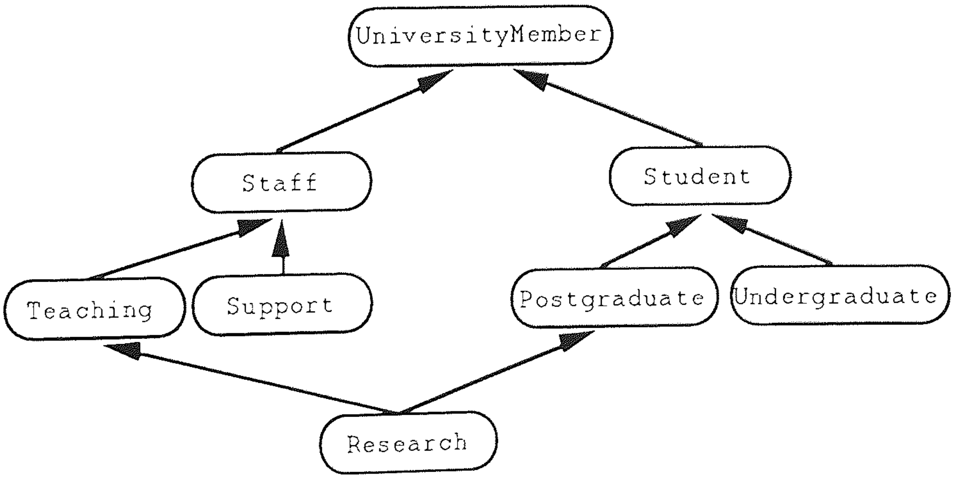


Figure 2.6 Example of multiple inheritance.

Single inheritance forces the choice of one superclass, when possibly several existing classes are equally suitable; each of the potential candidates is able to contribute some functionality to a new class. In this situation, single inheritance necessitates re-implementing in the new class the functionality of the excluded classes, thereby defeating the purpose of facilitating reuse through inheritance (Vlissides & Linton, 1988). Sakkinen (1989) observes that single inheritance can lead to unnaturally asymmetric constructions. Booch (1990) notes that the need for multiple inheritance is still a subject of great debate, but adds that his own experience indicates that although it is not always needed, multiple inheritance is useful.

Problems with Inheritance

Snyder (1986) and Micallef (1988) have discussed the potential conflict that arises between inheritance and encapsulation, stemming from the fact that a subclass has direct access to the internal structure of a superclass. This complicates the modification of a class implementation without adversely affecting descendant classes. The 'Law of Demeter' (Lieberherr & Riel, 1988) states that methods of a class should not depend on the structure of other classes, and these methods should also limit sending messages to objects of other classes. This guideline encourages the building of loosely coupled classes, and hence could help alleviate the problem of compromising encapsulation. Taenzer *et al* (Taenzer, Ganti & Podar, 1989) notice that when an object sends itself a message using `super` or `self` (in Smalltalk semantics), it could cause a search for the corresponding method up and down the inheritance hierarchy respectively. They liken the method matching process up and down an inheritance hierarchy to the motion of a yoyo (calling it the 'yoyo' problem), and believe that it increases implementation dependency between classes. Creating a new subclass or modifying an intermediate class in an inheritance chain, requires a study of the implementation of other classes to discover what messages an object sends itself and which need redefinition. Again encapsulation can be weakened by this dependency between classes.

Two problems prominent in multiple inheritance are *name collision* and *repeated inheritance*. Name collision occurs when more than one superclass of a class uses the same identifier for an instance variable or method (Figure 2.7(a)). Three approaches are taken by different object-oriented languages to resolve this conflict. Both Smalltalk and Eiffel regard a name clash as illegal, and compilation of the new class would therefore fail. Eiffel also allows renaming of the offending elements (instance variables or methods) to remove confusion. It is also possible for the language semantics to assume that the same name used in different superclasses refers to the same element; this approach is taken by CLOS. C++ on the other hand, permits a name clash, but requires that all references to the name must be fully qualified in relation to its source of declaration.

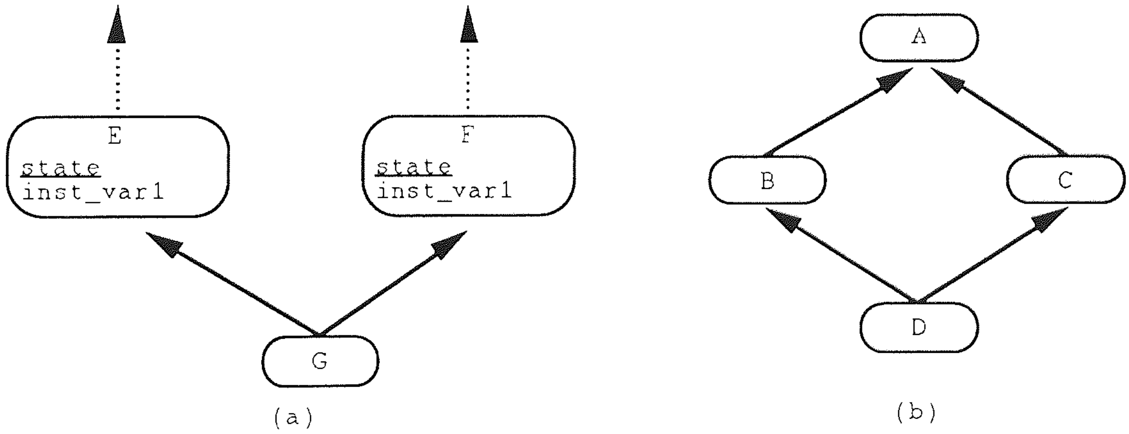


Figure 2.7 Problems in multiple inheritance with (a) name collision and (b) repeated inheritance.

Repeated inheritance is the scenario when "a class is an ancestor of another in more than one way" (Meyer, 1988) (Figure 2.7(b)). Different solutions to this problem are used in different languages. In Smalltalk and Eiffel, repeated inheritance is regarded as illegal. Eiffel, again, also allows renaming to remedy this. One technique in C++ is to use fully qualified names to distinguish which specific copy of an element is being referred to. C++ also uses the policy that multiple references to the same superclass in fact denote the same class. This approach is taken when the repeated superclass is defined as a *virtual base class*. (In C++, a virtual base class is designed solely to be shared by other classes, and cannot be instantiated; it is similar to an abstract class in Smalltalk.) CLOS uses a mechanism known as a *class precedence list*. The list is calculated each time a new class is added, which includes the new class and all its superclasses without duplication. This list is constructed based on two rules: first, the new class precedes its superclasses; second, each class sets the precedence order of its direct superclasses. This list flattens the inheritance graph, and inheritance is handled using single inheritance. A new class is only allowed if its complete precedence ordering can be worked out.

Inheritance and Genericity

Genericity is the ability to parameterise a software element with one or more types. For example, consider a generic swap procedure, which can be parameterised with the type of values it handles; this procedure can be defined in Ada (Watt, Wichmann & Findlay, 1987) as in Figure 2.8:

```
generic
      type Item is private;
procedure Swap (X, Y : in out Item) is
      Old_X: Item := X;
begin
      X := Y; Y := Old_X;
end Swap;
```

Figure 2.8 A generic definition in Ada.

`Item` is a generic type. `Swap` is a procedure template which must be instantiated by supplying an actual type parameter for `Item` when the procedure is to be used. `Swap` can be instantiated by the types `Integer` and `Character`, say, as follows:

```
procedure Swap_Int is new Swap(Integer);
procedure Swap_Char is new Swap(Character);
```

`Swap_Int` and `Swap_Char` can now be used to swap pairs of integer and characters respectively.

Meyer (1986) refers to this form of genericity as *unconstrained*, since there is no restriction on the actual type that may be used for instantiation. He differentiates this with *constrained* genericity, where a generic definition will be meaningful only if the actual type parameter satisfies some conditions. Consider writing a generic function which returns the minimum of two values (Figure 2.9):

```
generic
   type Item is private;
function Minimum (X, Y : Item) return Item is
begin
      if X <= Y then return X; else return Y end if;
end Minimum;
```

Figure 2.9 Generic function in Ada.

Such a function is only meaningful if it is instantiated with types for which the comparison operator "<=" is defined. So the generic part of the Ada definition must be modified as follows (Figure 2.10):

```
generic
type Item is private;
with function "<=" (A, B : Item) return BOOLEAN is <>;
function Minimum (X, Y : Item) return Item;
```

Figure 2.10 Constrained genericity.

The `with` clause is used to introduce generic formal parameters which are subprograms.

Both forms of genericity can be simulated using inheritance (Meyer, 1986). The idea is to associate a class with each formal generic type parameter; for constrained genericity, the constraining operations of the formal generic type parameter become instance methods of the class. This class is then used as an abstract superclass of the classes for the actual types, each subclass implementing its own methods for the constraining operations. A simple way to simulate inheritance in languages like Ada and Algol 68 is overloading, where the same subprogram name may be used for operations for different types. "However, this solution falls short of providing true polymorphic entities as in languages with inheritance, where ... an operation will be carried out differently depending on the particular form of an entity at run-time ..." (Meyer, 1986). A better alternative to overloading in languages like Ada and Modula-2 (Koffman, 1988) is to use a variant record type to include all the relevant types. Each operation would then use a case statement to discriminate between the types and carry out the appropriate task in each case; an example is shown in Figure 2.12. "Such a solution, however, is unacceptable from a software engineering point of view: it runs contrary to the criteria of extensibility, reusability and compatibility" (Meyer, 1986). Each time a new type is added, modification is required on the variant record type declaration, and the case statement in every operation will also need to be changed to take this new type into consideration. Both Meyer (1988) and Blair *et al* (1989) agree that inheritance is the more powerful mechanism.

Inheritance and Delegation

Wegner (1987) defines *delegation* as a mechanism that allows objects to delegate responsibility for performing an operation or finding a value to one or more designated "ancestors" or *prototypes* (they have also been referred to as *exemplars*) (Lieberman, 1986). For example, an object `A` wishes to print its representation on a text stream `S`. It may not know how to manipulate the stream directly, so it delegates the task to an object `printing` which interfaces with the stream (Figure 2.11).

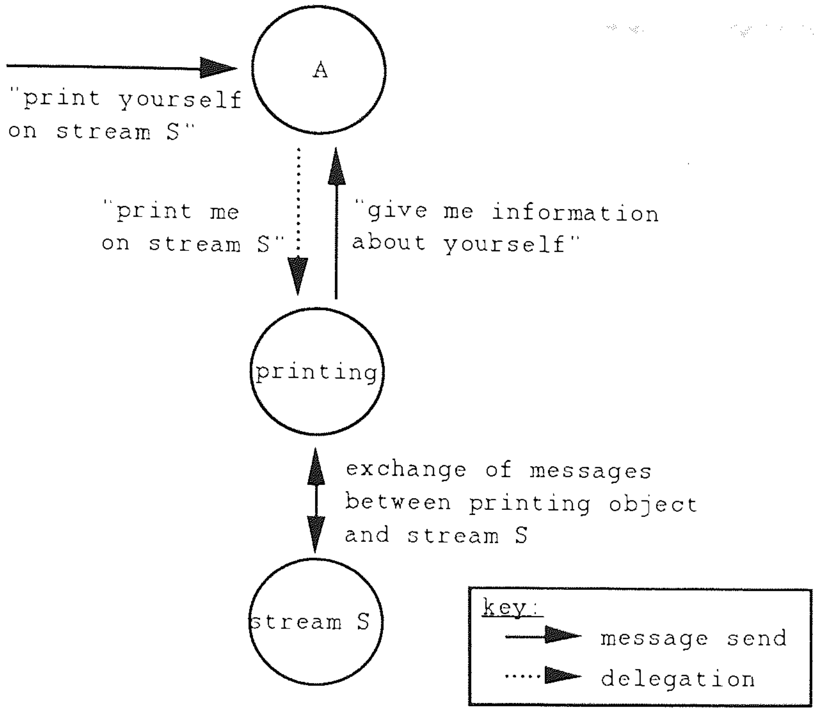


Figure 2.11 Delegation: object A delegating to object printing.

The object printing needs information about the state of its *client* or *customer* (object A) to complete its job, so it sends the appropriate messages back to object A to obtain this. When a client first delegates to a prototype, a reference to itself (the client) is implicitly passed to the prototype, so it is possible for the prototype to send messages back to the client.

In delegation, dynamic sharing is realised in an instance hierarchy rather than in a class hierarchy; delegation and class are therefore orthogonal concepts because the definition of delegation is independent of the class notion. The pattern of delegation varies dynamically as each object has the decision when and where to delegate during execution; this contrasts with inheritance where the pattern is fixed once classes are added to the inheritance structure (Wolczko, 1992). Lieberman (1986) considers that this makes delegation more flexible, and therefore delegation is a more powerful way of organising objects. He states that delegation can be used to model class-based inheritance whereas the converse is not possible, principally because it is impossible to make an instance depend on another in inheritance. Wolczko (1992), on the other hand, sees a direct analogy between delegation and inheritance: each superclass is viewed as defining a component object to an instance of a subclass, rather like an extended part of the instance. A message sent to an instance itself using `super` delegates the task to a component object of the instance, and `self` in a superclass method represents a reference to the customer object, just like a prototype sending a message back to its client in delegation.

Stein (1987) disagrees that delegation is necessarily more powerful than inheritance. He suggests that delegation can be simulated with inheritance by mapping prototypes into classes. This is facilitated because classes can be treated as objects. A class inherits class variables from its superclasses, as well as their values for these variables; this is similar to delegation where a client shares the attribute values of its prototypes. However, for languages in which class are not treated as objects (such as C++ and Eiffel), Stein's (1987) suggestion is achievable.

2.6 Polymorphism and Dynamic Binding

Cardelli & Wegner (1985) state that "conventional typed languages, such as Pascal, are based on the idea that functions and procedures, and hence operands, have a unique type. Such languages are said to be monomorphic, in the sense that any value and variables can be interpreted to be of one and only one type. Monomorphic programming languages may be contrasted with polymorphic languages in which some values and variables may have more than one type". They identify two main types of polymorphism: *ad-hoc* polymorphism and *universal* polymorphism. Ad-hoc polymorphism includes *operator overloading*, a mechanism used in procedural languages where semantically unrelated operations happen to share the same name, which map to different implementations depending on the number and types of arguments used in different contexts. An example of operator overloading in Modula-

2 is the use of the "+" symbol for both integer addition and set union. *Coercion* is also a form of ad-hoc polymorphism, where "operations can handle input of mixed types" (Graham, 1991). An example of coercion in C (Kernighan & Ritchie, 1978) takes place in the expression "`aFloat + anInteger`", where the operator "+" takes in an integer and a floating-point number; the integer is automatically converted to a floating-point number before the expression is evaluated.

Universal polymorphism comprises *parametric* polymorphism and *inclusion* polymorphism. Parametric polymorphism involves modules which can be called with actual parameters from a range of types. Genericity is an example of parametric polymorphism. Inclusion polymorphism is enabled by the inheritance mechanism present in object-oriented languages. This form of polymorphism is concerned with the ability to send the same messages to objects of different classes, where the classes are related by inheritance; instances of different subclasses can respond to the same messages found in their common superclass. Blair *et al* (1989) also mention *operational* polymorphism in association with object-oriented systems, where methods with the same message name can coexist in classes completely unrelated to each other by inheritance. Operational polymorphism, where message names are overloaded, is facilitated by strong encapsulation.

Several authors believe that *dynamic binding* is a necessary element of object-oriented programming (Pascoe, 1986; Cook, 1986; Cox, 1984; Booch, 1990). Booch also states that dynamic binding and polymorphism go hand in hand. With static binding, environment code is explicitly dependent on the classes known when the code was developed and compiled; any change when a new class is introduced potentially ripples through the entire environment, requiring modifications and recompilation. Consider the following Modula-2 code fragment (Figure 2.12) in which a procedure `DrawPicture` is defined which imports an array of shapes (`Square`, `Triangle` or `Circle`) and draws each in turn:

```
MODULE Picture;
TYPE ShapeType = (Square, Triangle, Circle);
          Shape = RECORD
                     CASE Kind: ShapeType OF
                       Square : Origin : Point; Length : INTEGER |
                       Triangle : Point1, Point2, Point3 : INTEGER|
                       Circle : Centre : Point; Radius : INTEGER
                     END (* CASE *)
                     END; (* RECORD *)
           APicture = ARRAY [1..10] OF Shape;
PROCEDURE DrawPicture (ThisPicture: APicture);
VAR Index : [1..10];
BEGIN
          FOR Index := 1 TO 10 DO
             WITH ThisPicture[Index] DO
               CASE Kind OF
                  Square : DrawSquare(ThisPicture[Index])|
                  Triangle : DrawTriangle(ThisPicture[Index])|
                  Circle : DrawCircle(ThisPicture[Index]
               END (* CASE *)
            END (* WITH *)
          END (* FOR *)
END DrawPicture;
```

Figure 2.12 Example of static binding in Modula-2.

The types of shape known at compile time are listed in the enumerated type ShapeType and are fixed. DrawPicture can only differentiate between these known types at run time, using the Case statement. Whenever a new type of shape (e.g., Hexagon) is to be add to the module, the enumerated type ShapeType and the variant record Shape will have to be modified; and so will the Case statement in every procedure dependent on the variant record type Shape, such as DrawPicture. This can be both time consuming and error-prone.

Re-implementing the same example in Smalltalk-80, which supports polymorphism and dynamic binding, DrawPicture can be expressed as:

```
drawPicture: thisPicture
     1 to: thisPicture size do:
        [:index | (thisPicture at: index) draw]
```

where draw is an instance method in Shape and overridden in each of its subclasses to provide the appropriate drawing sequence. drawPicture: thisPicture is itself an instance method in a class, which takes in an array of shapes. When a new class of shape is to be added to the application, no change to `drawPicture: thisPicture` is necessary, since every class of shape knows how to draw itself at run time. Polymorphism ensures that `draw` can be sent to all objects in `thisPicture` regardless of the classes they belong to, and dynamic binding puts no limit on the class of each object in `thisPicture` at compile time.

With dynamic binding, environment code which depends on the classes of objects (e.g., `drawPicture`) can be reused without change. This is made possible by polymorphism (inclusion and operational); the classes of objects are irrelevant to the environment code as the same messages can elicit the appropriate responses from different objects depending on their classes, but this fact is invisible to the environment code that makes use of these objects. Polymorphism "is desirable because it enables us to write extremely general-purpose programs in a transparent manner — the bare algorithm and no frills" (Harland, 1984).

2.7 Summary

Winblad *et al* (1990) state that the basic mechanisms of the object-oriented paradigm comprise objects and classes, messages and methods, and inheritance, and the key concepts of object orientation include abstraction, encapsulation, polymorphism, and persistence. Objects are instances of classes, which define the states and implement the protocols of similar objects. Objects represent encapsulations of data abstractions, each object owning a unique identity. Communications between objects occur via message sending; when an object receives a message it executes the method of the message. Inheritance between classes in an inheritance structure facilitates code reuse, and eases maintenance through code factoring. Polymorphism with dynamic binding enable the writing of robust and flexible code which is resilient to changes.

Several current research topics such as *persistence* (Morrison, Brown, Carrick, Connor, Dearle & Atkinson, 1987; Thatte, 1986) and concurrency in object-oriented programming (Yonezawa & Tokoro, 1987) have not been discussed as they are not central to this research. Persistence in the object-oriented paradigm refers to the permanence of an object, i.e., the amount of time for which it is allocated space and remains accessible in the computer's memory. Objects in a persistent system can outlive the programs in which they have been created, as long as they can be referenced (Lewis, 1991). The *virtual image* of Smalltalk, in which objects exist, offers a limited form of persistence; saving the virtual image between sessions preserves the states of objects. Object-oriented database management systems such as Gemstone (Maier, Stein, Otis & Purdy, 1986) provide the ability to store and share objects in a multi-user environment.

Lim & Johnson (1989) state that "... OOP can alleviate the concurrency problem for the majority of programmers by hiding the concurrency inside reusable abstractions". The usual approach in concurrent object-oriented systems is to provide objects with independent threads of control; each object's thread of control conceptually executes concurrently with the object's methods, and the threads of other objects (Atkinson, Goldsack, Di Maio & Bayan, 1991). Objects possessing such threads are termed *active*, while those that do not are termed *passive*. Concurrent object-oriented programming languages such as ABCL/1 (Yonezawa, Briot & Shibayama, 1986) provide mechanisms for active objects and synchronisation. Smalltalk and Ada also support multitasking by providing *process* and *task* respectively, while C++ uses the Unix system call *fork* to implement concurrent objects.

Chapter 3 The Software Life Cycle and Object-oriented Development

3.1 The Software Development Life Cycle

In software engineering, the *software life cycle* is an overall organizational framework describing the activities of the development, use and maintenance of a software system. Three main phases can be identified in the software life cycle: *analysis*, *design*, and *implementation*. The number of subdivisions in each of these phases varies between different authors. The order of the stages involved in the life cycle and the transition criteria from one stage to the next depend on the underlying model on which the life cycle is based (Boehm, 1988).

3.1.1 The Waterfall Model

The traditional model for the software life cycle is the *waterfall model* (Figure 3.1) (Sommerville, 1992) which "attempts to discretize the identifiable activities within the software development process as a linear series of actions, each of which must be completed before the next is commenced" (Henderson-Sellers & Edwards, 1990). Subsequent improvements to the waterfall model recognised the importance of feedback loops between stages. However, the feedback loops in this model are confined only "to successive stages to minimize expensive rework involved in feedback across many stages" (Boehm, 1988); the intent of the model is still "to proceed forward from each phase to the next" (Wasserman & Pircher, 1991).

Problems with the Waterfall Model

The main criticism of the waterfall model is that it does not recognise the role of iteration in the software development process (Sommerville, 1992), due to its emphasis on a purely sequential flow between the stages and activities of the software life cycle. Booch (1991) observes that the waterfall model "is sometimes treated as a sacred, immutable process in which work blindly flows down from one phase to another ... that products from an early phase are written in granite, which then serve as the costly-to change input of a later phase". However, as Ratcliff (1987) notes: "software development is highly iterative and parallel in nature and does not proceed in a single, clear-cut sequence".

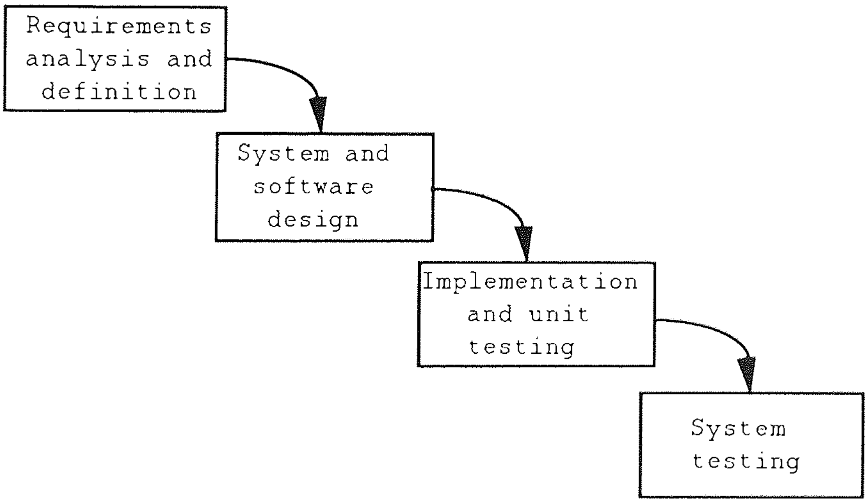

Requirements analysis and definition
System and software design
Implementation and unit testing
System testing

Figure 3.1 The waterfall model (Sommerville, 1992).

The waterfall model is hence unrepresentative of the activities which actually go on during the software process (McCracken & Jackson, 1982; Gladden, 1982); Hatley & Pirbhai (1988) suggest that "this view obscures the true nature of systems development: it has always been an iterative process in which any given step can feed back and modify decisions made in a preceding one". In particular, Booch (1991) states that the inherent incremental and iterative nature of object-oriented design, describing it as *round trip gestalt design*, makes it orthogonal to the traditional waterfall life cycle approach to software development.

Another source of problem with the waterfall model is "its emphasis on fully elaborated documents as completion criteria for early requirements and design phases ... However, it does not work well for many classes of software" (Boehm, 1988). This expectation is only realistic in some classes of software such as compilers and secure operating systems (Boehm, 1988) where a finite and fixed set of requirement and functions can be determined; however, it has difficulty accommodating the natural uncertainty that exists at the beginning of many projects.

More importantly, the classical waterfall model is not efficacious when used for object-oriented software development, as Korson & McGregor (1990) note: "Problems with traditional development using the classical life cycle include no iteration, no emphasis on reuse, and no unifying model to integrate the phases". Without reuse, which is a major feature of the object-oriented paradigm, each system has to be built from scratch instead of utilising existing tried and tested classes, hence possibly increasing maintenance costs (see [2.4, 2.5]). Coad & Yourdon (1990) point out that the difference in point of view between using data flows of structured analysis and using structure charts/top-down functional decomposition of structured design, may cause problems by introducing an unnecessary boundary between analysis and design in the software life cycle resulting from a shift from the problem domain in analysis to the solution domain in design; on the other hand, the "analysis and design of the traditional life cycle, while remaining separate activities in the object-oriented life cycle, work together closely to develop a model of the problem domain".

The software life cycle for object-oriented development eliminates the distinct boundaries between the various phases in the life cycle (Meyer, 1989a; Coad & Yourdon, 1990). The primary reason for this blurring of boundaries is that "the items of interest in each phase are the same: objects" (Korson & McGregor, 1990). Objects and their interactions are identified in both analysis and design. Both object-oriented analysis and design view the problem domain as a set of interacting objects. Information derived in an object-oriented analysis phase becomes an integral part of the object-oriented design rather than merely providing input to the latter as with structured analysis and structured design. This smooth and seamless interface between phases is facilitated by the homogeneity of their conceptual framework, namely objects and classes. Korson & MacGregor (1990) also state that an object-oriented analysis and object-oriented design approach is very natural and flexible "... in the sense that the design pieces are closely identified with the real-world concepts which they model ... quickly adapting to changes in the problem specifications" (Korson & MacGregor, 1990).

3.1.2 Object-oriented Development Cycle

The Fountain Model

Henderson-Sellers & Edwards (1990) describe an alternative to the waterfall model for the software life cycle, the *fountain model*, which reflects the stages and activities of object-oriented software development more closely, emphasising the significant amount of iteration and overlap that exists (Malhotra, Thomas, Carroll & Miller, 1980; Meyer, 1989; Turner, 1987). The diagrammatic representation of the fountain model (Figure 3.2) shows clearly its difference with the waterfall model, with the overlap and iteration between stages being emphasised. The software life cycle "grows upward to a pinnacle of software use, falling only in terms of necessary maintenance" (Henderson-Sellers & Edwards, 1990), reverting to a previous level to begin the climb again — hence the name of the model. This repetition of activities also occurs throughout the lower-level stages.

An object-oriented system is developed essentially as a system of interacting classes, where each class can usually be developed independently; the stages of the life cycle model can, therefore, be applied accurately to the development cycle of individual classes (Gindre & Sada, 1989) (Figure 3.3). During the explicit stages of generalization and aggregation in the fountain model, application-specific classes are revised so that they may be sufficiently generic to be useful to other applications. These extra stages of generalization and aggregation require a greater effort than one-off design and implementation in the short term, but in the long term, this effort can lead to significant reduction in overall system development time and effort. This reduction is facilitated by the emphasis on code reuse in object-oriented development, where bottom-up development of new classes based on existing classes often occurs within the framework of an overall top-down object-oriented system analysis and high-level design. Hence object-oriented development is a combination of both top-down system development and bottom-up development of classes. This facilitates modifications as changes can be made interactively between class development and system specification; there is therefore no longer a need to freeze the overall system requirements specification at an early stage of the system life cycle (Henderson-Sellers & Edward, 1990).

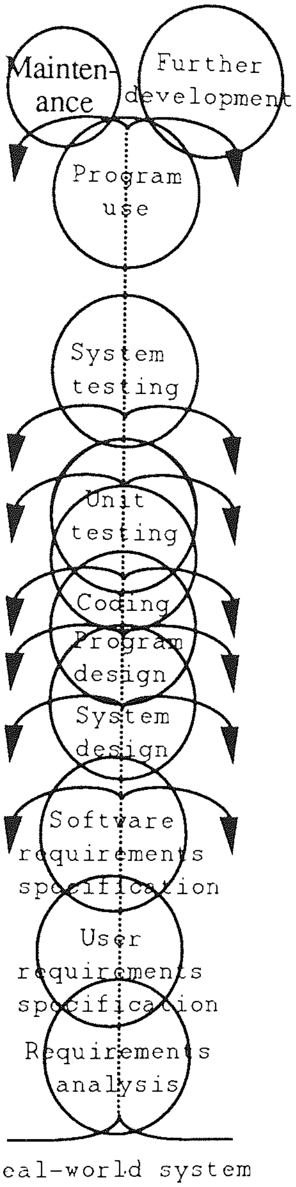


Figure 3.2 The fountain model for system development.

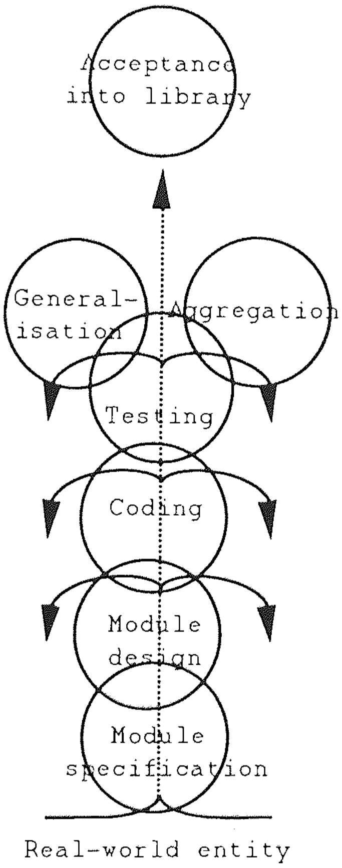


Figure 3.3 The fountain model applied to the development of individual or a cluster of classes.

The Cluster Model

Often a small *cluster* of conceptually tightly related classes can be developed together. Meyer (1989) describes the *cluster model* which relates to the life cycle of individual clusters of classes. The cluster model consists of three phases: first, a specification is written for a cluster of classes; next, this specification is design and implemented; finally the classes in the cluster are validated and generalised. Again, generalisation is an extra development step carried out to improve the generality, genericity and robustness of the classes so that they may be useful to future development projects. Reuse is facilitated, thereby offsetting any short-term development overhead. The cluster model life cycles of different clusters are independent, occurring at different times and concurrently (Figure 3.4); within the cluster model, the stages of the fountain model for the life cycle of a class (Figure 3.3) can hence apply concurrently to the classes in a cluster.

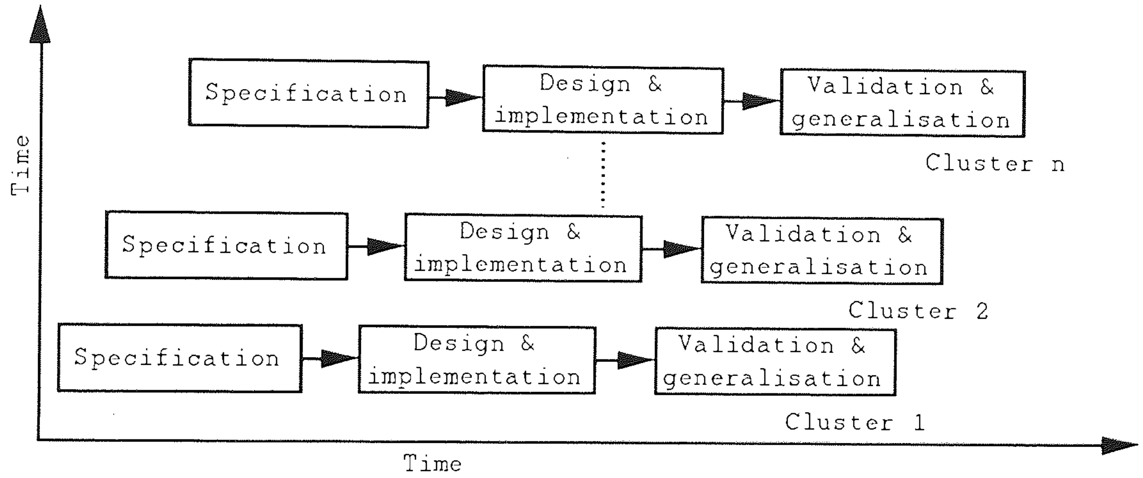


Figure 3.4 Cluster model of different clusters occurring concurrently.

The cluster model can be incorporated in the fountain model of a system at the program design stage; requirements, design and implementation stages of the fountain model of a system can progress and iterate over time while individual classes or clusters of classes undergo their own cluster and fountain life cycle. The advantage of this approach is that "characteristics of a system, which evolves dynamically as user's and analyst's knowledge grow, can be incorporated in the overall life-cycle model" (Henderson-Sellers & Edwards, 1990). System requirements and design spawn clusters which are passed on to programmers for detailed design and implementation. These implemented clusters are later incorporated into the final system, while system requirements analysis and system design iterate. A change in requirements may cause modifications to or even the abandonment of a single cluster but this will not cause major redesign of the system since the overall system synthesis occurs later in the system's life cycle. On the other hand, changing the requirements of a system designed and implemented using traditional top-down methods, could result in a great deal of reworking on the detailed design which has been decomposed from the top-level design and therefore highly dependent on it.

The Spiral Model

Wasserman & Pircher (1991, 1991a) use the spiral model of Boehm (1988) to describe the development process of an object-oriented system. Like the fountain model this model emphasises the continual, iterative elaboration of the details of classes, objects and their relationships, rather than the clear transitions between analysis, design and implementation found in the waterfall model. Another similarity this spiral model has with the fountain model is that it also stresses the identification of reusable classes and objects, and the derivation of new classes from existing ones.

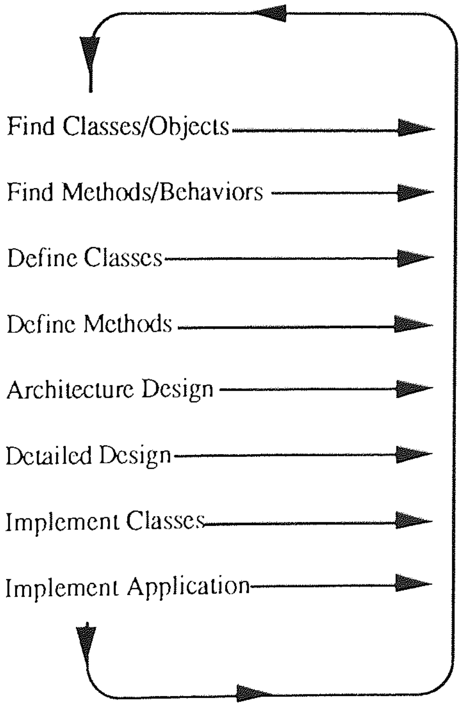


Figure 3.5 Spiral model for object-oriented software development (Wasserman & Pircher, 1991).

3.2 Object-oriented Analysis

Several techniques have been proposed by various authors for the analysis phase of object-oriented development; this section looks at some of these techniques. The first two are object-oriented approaches to the analysis process; here influence from relational databases is apparent. Next, methods for modeling the problem domain based on using informal English and the traditional Structured Analysis are reviewed.

Object-oriented System Analysis: Modeling the World in Data

Shlaer & Mellor (1988) propose an object-oriented analysis technique which they call Object-oriented System Analysis: Modeling the World in Data. It uses a graphical notation called an *information structure diagram* based on various forms of *entity relationship diagram* used by other authors (Tsichritzis & Lochovsky, 1982; Martin, 1985; Chen, 1976). The method starts by constructing an *information* (or data) *model* showing objects, their attributes, and the relationships between the objects. In Shlaer & Mellor's terminology, an object refers to a class. Attributes are classified into *naming*, *referential* and *descriptive* (Figure 3.6 (a)). Naming attributes are designated with *, o indicates a descriptive attribute and (r) denotes a referential attribute. Naming attributes constitute the identities of instances of an object; referential attributes carry values which relate an instance of an object to an instance of another object (e.g., see Figure 3.6 (b), where the student object may contain an attribute attends which relates it to the university object), while descriptive attributes provide facts about instances of an object. The potential problem with the use of (naming) attributes to denote identities of object instances has already been explained in Section 2.2. The requirement that attribute values must be atomic, e.g., integer, string, etc., (evidence of influence from database technology) would seem to be too limiting for building an object-oriented model of the problem domain where complex objects have to be modeled. Relationships between objects are modeled in terms of cardinality (or multiplicity) — one-to-one, one-to-many or many-to-many (Figure 3.6 (a)) — and whether a relationship is conditional (or modality (Graham, 1991)). A conditional relationship between objects is one in which not necessarily all instances of an object involved participate in the relationship. Anderson (1990), however, expresses concern about the ease with which these relational database issues could be implemented in object-oriented languages like Smalltalk or C++.

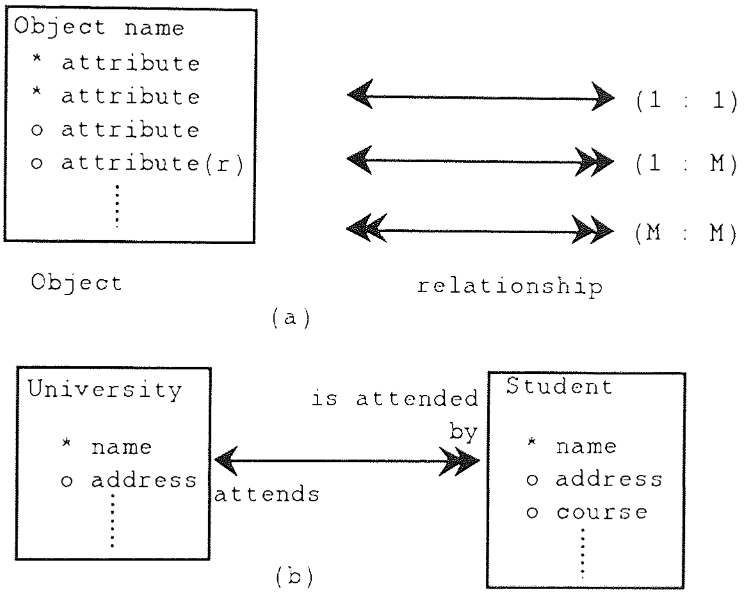


Figure 3.6 (a) Symbols for an object and object relationships and (b) an example information structure diagram.

Shlaer & Mellor (1988) consider the task of identifying objects as simple, starting by looking for candidates in five categories: tangible things (e.g., plane, car, book); roles (e.g., lecturer, student); incidents (e.g., flight, accident); interactions (e.g., loan, lecture); specifications (e.g., course, loan type). Objects can be organised into a hierarchy of supertypes and subtypes, where a supertype contains attributes shared with its subtypes (Figure 3.7); this hierarchy may be used to represent inheritance relationships amongst classes.

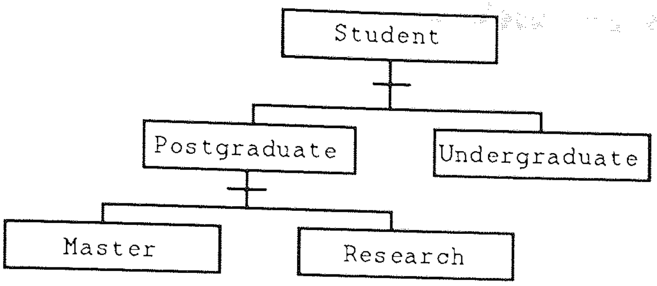


Figure 3.7 Supertype/subtype hierarchy used by Shlaer & Mellor.

From the information model, life cycles (*state models*) of objects are constructed using state transition diagrams which record the behaviour over time of instances of each object. Each state has some associated actions which an instance must carry out when it arrives at that state; this differs from the use of state transition diagrams by other authors where actions are more commonly associated with transitions (de Champeaux & Faure, 1992). Lastly, data flow diagrams (DeMarco, 1977) are used as *processing models* to capture the flow of data between processes (or states) in the state models. However, Coad & Yourdon (1990) point out that Shlaer & Mellor's method does not provide any mechanism for expressing the concepts of messages and methods; some authors (Winblad, Edwards & King, 1990; Graham, 1991) also note that there is no emphasis in the method on the encapsulation of processing with data in an object.

Object-oriented Analysis

Coad & Yourdon (1990) describe an object-oriented analysis method which consists of five layers (stages): Subjects, Objects, Structures, Attributes, and Services — or 'SOSAS'. Like Shlaer & Mellor (1988), the term 'object' is used in place of 'class'. Coad & Yourdon state that "real systems have a substantial number of Objects and Structures", subjects provide the mechanism for partitioning the overall model of a complex problem area into sub-models of manageable size, usually between five to nine objects — these numbers being derived from Miller's (1956) work described in his paper entitled: "The magical number seven, plus or minus two: Some units on our capacity for processing information". Subjects thus serve to control the proportion of a model a reader is able to consider and comprehend at one time, and they also serve as a guide to the diagrams in the object-oriented analysis model. Subjects correspond to Booch's concept of *class categories* (1991) and to Meyer's *cluster* concept (1987). In the subject layer, a subject is initially assigned to each object and to each structure (either a parent-child structure or inheritance structure between classes). When instance and message connections between objects and structures are identified during the subsequent attribute and service steps, tightly coupled subjects are merged until eventually a small number of subjects remain. Subjects are represented in a subject layer diagram with connections denoting potential message interactions between the classes of the subjects (Figure 3.8). Diagrams of the subsequent layers may be partitioned according to their subjects.

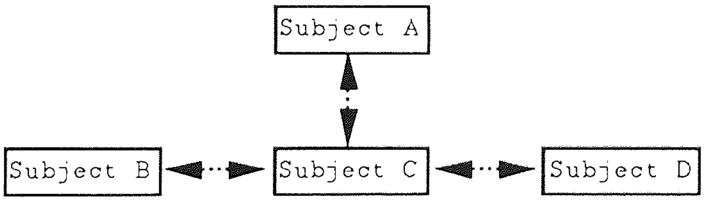


Figure 3.8 Coad & Yourdon's subject layer diagram.

In the object layer, objects are identified in detail within each subject area and recorded in an object layer diagram. Coad & Yourdon provide a list of possible sources of objects, including structures (e.g., "kind of" and "part of" relationships), other systems and devices, events (i.e., historical events that must be recorded, e.g., a bank loan), roles (roles users play interacting with the system, e.g., bank customer, bank clerk), locations (physical locations or sites important to the system), organisational units (groups users belong to). Looking for nouns in a written description of the problem is also suggested as a strategy (this approach is discussed in detail by Abbott (1983) and is outlined in a subsequent section) .

In the structure layer, two types of structure are identified: *classification structure* and *assembly structure*. The classification structure uses the inheritance concept and shows generalisation and specialisation of real world entities (Figure 3.9 (a)). Multiple inheritance is allowed (Coad & Yourdon, 1991) but the only provision for conflict resolution is to annotate each attribute and method explicitly (Graham, 1991). The assembly structure deals with the composition structure (or aggregation) of real world objects, and can be related to the "parent-child" hierarchy and "containing relationship" described in Section 2.2 (Figure 3.9 (b) shows an example assembly structure). Three types of composition are distinguished: part-whole, container-contents and collection-members. Graham (1991) considers the last two types of composition relationship to be merely extra structure and not special cases of composition, as contents and members can be regarded as parts of containers and collections (which are the wholes) respectively.

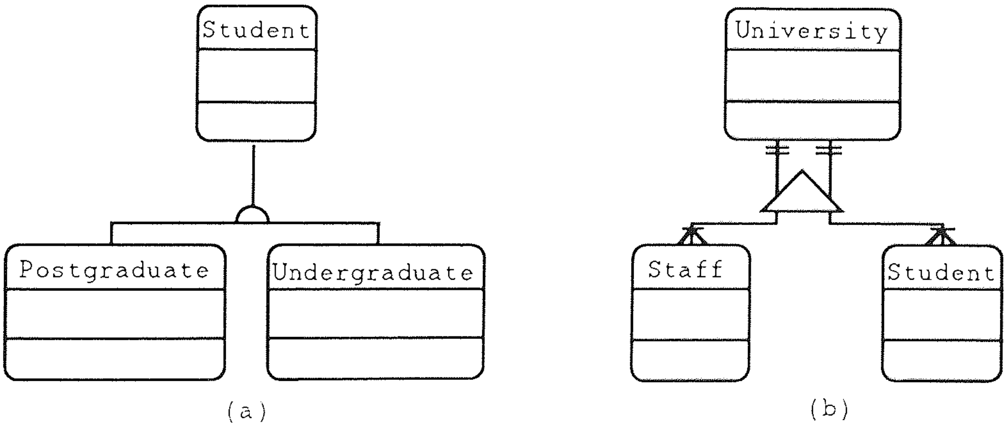


Figure 3.9 (a) An example object inheritance structure and (b) an example assembly structure.

In the attribute layer, attributes which describe instances of an object are determined and noted in the object's symbol, e.g., Student, Postgraduate, Undergraduate in Figure 3.9 (a). Again, the influence of relational databases is apparent here as the authors require that attributes have to be atomic, as with Shlaer & Mellor's approach. Also, like Shlaer & Mellor's method, attributes are used to identify instances, which can cause problems (see Section 2.2). Next, relationships between instances of different objects (Figure 3.10 (a)) are considered and represented via instance connections, where "an instance connection is a mapping from one instance to another"; each instance connection also implies a corresponding message connection between the instances involved (Figure 3.10 (b)). For each instance connection, its multiplicity and modality are considered; this information is also included in an assembly structure. Instance and message connections model the "data semantics" (Graham, 1991) of instances. Graham (1991) suggests that it may be useful also to consider and represent the data semantics of objects (classes) as relationships and messages could also arise between objects (i.e., classes).

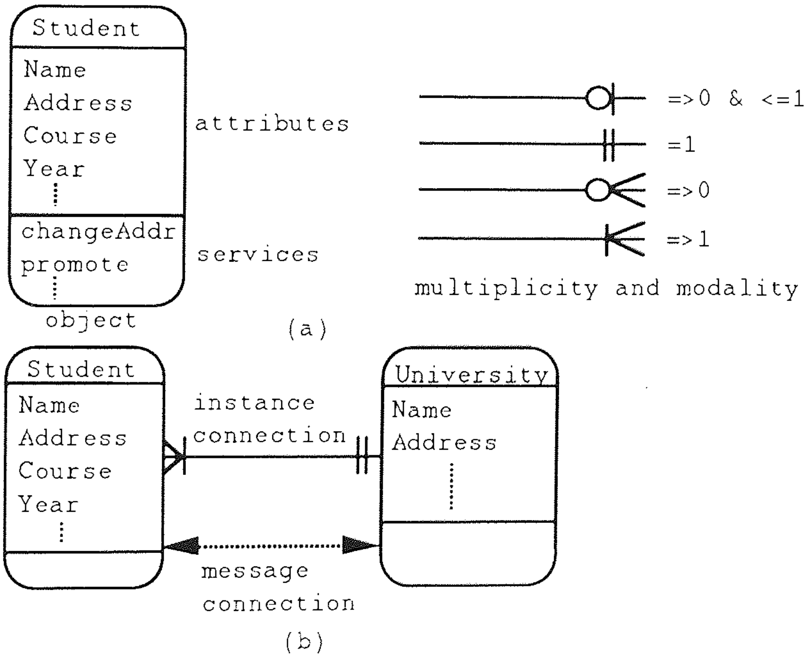


Figure 3.10 (a) Object symbols with attribute and service sections, and the different combinations of multiplicity and modality in instance connections; (b) instance connection and message connection.

The final stage of Coad & Yourdon's method, the service layer, concentrates on describing the functional requirements of the system. A service in an object corresponds to a message in the protocol of a class. Each object must provide basic services for adding, changing, deleting instances, as well as services characterising the object's behaviour. Examination of object life histories and state-event-responses are strategies suggested for identifying services an object has to provide. Message connections between object instances represent message-sending relationships between instances. Enumeration of message names with each message connection is allowed, although it could result in very confusing diagrams; moreover, it is redundant as the services named clearly indicate which messages may be passed; Graham (1991) agrees with this observation. Finally, the services are specified using template forms based on the formal specification language Ina Jo described by Wing & Nixon (1989); where required, diagrams are also used for specifying services, including data flow diagrams, state transition diagrams, and decision trees.

Graham (1991) extends the object-oriented analysis method of Coad & Yourdon to include an extra stage, called "declarative semantics", or "rules", during which functional semantics, global control and business rules are explored and recorded. He calls this method SOMA, or semantically rich object-oriented analysis. Control rules are declared in objects for handling conflict resolution in multiple inheritance, exception handling, default values for attributes, and demons (which are methods that activate automatically whenever their triggering events occur). Business rules are used to express "second order" information such as dependencies between attribute values, e.g., a dependency between a student's year of course and the number of books he is allowed to borrow from the library. Global pre- and post- conditions relating to all methods of an object are allowed, as well as those pertaining to individual methods. Control rules can be encapsulated within objects, allowing for local variations between objects, or global rules can be inherited from the top-level object. Graham (1991) regards the encapsulation of rules in objects as enhancing reusability and extensibility of the requirements specification. This extra feature introduced in SOMA is intended to be useful mainly for commercial systems development where a relational or deductive database with object-oriented features is the envisaged target environment. Hence the feature may not be as applicable if a design derived from such an analysis model is to be implemented in an object-oriented programming language.

Informal English

Abbott (1983) suggests a strategy for deriving candidate classes, objects and their operations from an informal English description of a problem solution. "The nouns and noun phrases in the informal strategy are good indicators of the objects and their classifications", and an initial identification of the operations of these objects and classes may be based on "the verbs, attributes, predicates, and descriptive expressions in the informal strategy".

Abbott classifies nouns and nouns phrases into several categories: common nouns, proper nouns and direct references, mass nouns and units of measure. The difference between a common noun and a proper noun is that a common noun is a name of a class of individual entities (e.g., university), while a proper noun is the name of a specific entity or being (e.g., Aston University); like a proper noun, a direct reference refers to a specific entity, but without using its name (e.g., my university). Mass nouns are names of qualities, activities or substances, which individually are not regarded as entities (e.g., code, information) — a "good test for determining whether a noun is a mass noun being whether it can be used in the phrase "how much ..." " (e.g., "how much code ...", "how much information ..."). Units of measure refer to quantities of these qualities, activities and substances, and are usually names given to arbitrary units used for subdividing mass nouns (e.g., lines (of code), bit (of information)). Collective nouns which refer to groups of individuals, where each group itself may be considered an entity, and integral counts of common nouns, are also treated as units of measure. Based on the meaning of the various forms of noun, a common noun may be taken as an indication of a class, and a proper noun or a direct reference suggests an object (or an instance of a class).

Verbs which express acts or occurrences can be mapped to operations of the objects on which the acts are performed; for example, in the sentence "a bank customer deposits money in his account", the verb `deposits` could be selected as an operation of an `account` object. Attributes could be taken as operations on objects which return attribute values of the objects (e.g., `address` and `accountBalance` would be some of the (attribute) operations of a `bankCustomer` object), and predicates which designate properties or relations that are either true or false are usually represented as operations which test whether the predicates hold regarding the objects, returning true or false respectively (e.g., a `bankCustomer` object may receive the operation `inCredit`, which returns true or false depending on the value of the object's `accountBalance`). A descriptive expression characterises an object (e.g., "investment account customer"), and may be chosen as an operation which returns as its value objects fitting the description (e.g., a `bank` object may have an operation `investmentAccountCustomer` which returns a list of all its `bankCustomer` objects holding investment accounts).

The advantages of Abbott's approach include its simplicity, and it ability to help the user to concentrate on the vocabulary of the problem space (Booch, 1991). However, Abbott cautions that although at first glance the approach seems mechanical and lends itself to automation, "differentiating among types of nouns is a matter of semantics and not a simple syntactic distinction ... (which) requires knowledge of real-world phenomena and an understanding of the meaning of words". Often, the category that fits a particular noun depends on its use and not just on the noun itself. This concern with the difficulty in categorising nouns naturally also applies to differentiating different types of verb. Abbott's approach however, has been widely used by a number of authors and several object-oriented analysis and design methods, examples being HOOD (The HOOD Working Group, 1989), Sincovec & Wiener (1984) and CRC (Beck & Cunningham, 1989).

Structured Analysis

Another alternative to an object-oriented approach to analysis would be to use structured analysis techniques as a front end to object-oriented design. "This technique is appealing simply because a large number of analysts are skilled in structured analysis, and many CASE tools exist that support the automation of these methods" (Booch, 1991). Structured analysis techniques have been used by several authors for identifying objects in the problem domain. Alabiso (1988) suggests a set of rules for extracting possible objects and their methods from data flow diagrams and data dictionaries. A data flow diagram describes the functional requirements of the system in terms of the flow of data between processing. Alabiso's approach is based on the "essential model" proposed by Ward & Mellor (1986), which includes real-time extensions of the basic data flow diagram notation. Data in data flow diagrams and data names in a data dictionary are mapped to objects, while a data process may be assigned as a method of an input data object to the process. Figure 3.11 shows an example of this process where the `pop` process on the input data `aStack` is assigned as a method in the class `Stack`, of which `aStack` is an instance. Often output data is the same as input data, only slightly modified (e.g., `updatedStack` refers to the same instance `aStack` which has been modified after executing the `pop` method). Terminals in a data flow diagram represent entities external to the system which produce or consume data (also know as "sources" and "sinks"). Terminals can be mapped to objects, with added methods for extracting data from and delivering data to these objects. Data stores are files to which data is written, or from which data is read, by processes. They are treated similarly to terminals.

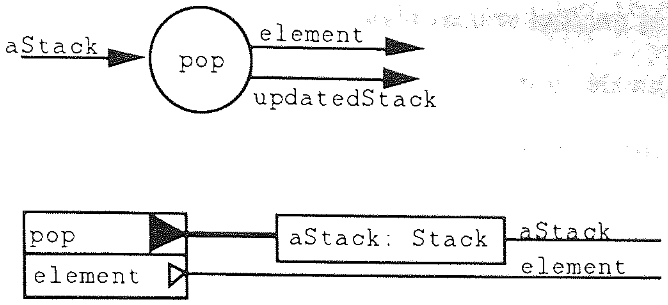


Figure 3.11 Example of Alabiso's approach to identifying objects and methods from a data flow diagram.

Functional decomposition of processes results in a hierarchy of data flow diagrams, where lower-level data flow diagrams describe processes in higher-level diagrams. This hierarchy of data flow diagrams is used to assist in the functional decomposition of methods identified; the detail of a method is provided by a lower-level data flow diagram of the process giving rise to the method. Entries in a data dictionary detailing decomposition of data elements in data flow diagrams are used to discover decomposition of corresponding objects identified. A data entry specifying a *sequence* in the data can be mapped to variables in the object (e.g., `a` is composed of `a1` and `a2` and `a3` implies that class `a` contains variables `a1`, `a2` and `a3`). *Repetition* in the data can mean the object contains a collection of other objects (e.g., `a` is composed of 1 to n `ax`'s means class `a` contains a set, list or array, etc., of instances of class `ax`). *Selection* in the data can indicate an inheritance relationship (e.g., `a` is composed of either `a1`, `a2`, or `a3`, may be interpreted as classes `a1`, `a2`, and `a3` are subclasses of class `a`).

Seidewitz & Stark (1986) use a technique called *abstraction analysis* to look for objects in data blow diagrams. The first step of abstraction analysis involves identifying a "central entity" which represents " the best abstraction for what the system does or models". Similar to *transform analysis* (Yourdon & Constantine, 1979) of Structured Design which looks for where input and output are most abstract in a data flow diagram, the central entity is located by determining a set of processes and data stores that are most abstract (this might require looking at lower-level data flow diagrams) and grouping these to form the central entity. Having determined the central entity in a data flow diagram, abstraction analysis proceeds by finding supporting entities by following data flows from the central entity and grouping process and data stores into abstract entities until all processes and data stores have been allocated. An *entity graph* is the result of the abstraction analysis process, and serves as the starting point for object identification. Often entities are mapped directly into objects; however, the authors concede that "identifying objects is not always this simple". Operations provided and used by each object are selected by examining each process of the corresponding entity for primitive processes, as well as looking at the data flows crossing boundaries between entities in a data flow diagram; each primitive process becomes a method. Lower-level data flow diagrams are used to repeat the process of object identification by again partitioning each data flow diagram into entities, based on how they support the top-level object's operations. This repetition continues until the lowest level data flow diagrams have been examined.

Booch (1991) states that "structured design, as normally coupled with structured analysis, is entirely orthogonal to the principles of object-oriented design". Alabiso (1988) concedes that the gap between the underlying models of structured analysis, based on the flow of data between processes, and object-oriented design, based on the object/message model, is significant, requiring "a sizable informal quantum jump" to derive an object-oriented design from data flow diagrams and data dictionaries. Coad & Yourdon observe that such a gap could be difficult to reconcile, especially when continual changes in requirements are difficult to move into the design with little support for traceability. Alabiso (1988) suggests that the transformation from structured analysis to object-oriented design would be better facilitated if structured analysis could be made "object aware"; that is, if "the Analysis Model itself were slightly modified to deal with 'objects, classes and methods', rather than 'data and processes', ... much less guesswork would be required to migrate from the Analysis Model to the Object Oriented Model".

A clear inadequacy in using structured analysis with object-oriented design is the lack of support for inheritance. Although Alabiso considers inheritance to be a purely design issue, Coad & Yourdon (1990) point out that the concept of inheritance corresponds to one of the basic methods humans use to manage complexity, and provides an important mechanism for the partitioning of a problem space; thus, its significance should not be neglected during analysis prior to object-oriented design. Inheritance is also a primary mechanism facilitating reuse in object-oriented development (Winder & Pun, 1988), which should be an important consideration even during analysis. Graham (1991) states that "inheritance and other structures [parent-child relationship and use relationship] are an important part of the domain semantics; the way objects are classified defines the domain, and often the purpose, of the application"; by considering inheritance "the structural features of the domain are revealed, including natural notions of specialization and generalization"(Graham, 1991).

3.3 Object-oriented Design

A proliferation of design methods for object-oriented development has emerged. Earlier methods such as HOOD (HOOD Working Group, 1989, 1989a) and the early work by Booch (1986) were heavily influenced by the ADA language, being designed primarily for ADA implementation. Structured Design (Yourdon & Constantine, 1979) has also been used as a basis for object-oriented design methods, e.g., OOSD (Wasserman *et al*, 1989, 1990). Recent work attempts to depart from these two approaches. The following outlines some of these different methods.

Object-oriented Structured Design (OOSD)

Wasserman *et al* (1989; 1990) propose a method for architectural design called Object-oriented Structured Design which aims to combined ideas and notation from Structured Design as described by Stevens *et al* (1974) (such as modules and structure chart) with object-oriented concepts such as abstract data types, class hierarchy, and inheritance. The method includes a design notation which bears influence from Booch's (1983, 1986, 1990, 1991) representation for ADA packages and tasks; the authors of OOSD describe their notation as "a superset of structure charts and Booch's notation". The main reason for using Structured Design as the basis for the method's notation stems from the authors' desire to build on notation and concepts (e.g., modularity) already familiar to most software designers, and to support a variety of analysis methods and target implementations. Designers could continue to use familiar concepts from Structured Design to document existing designs, but also make use of the new object-oriented features for designing object-oriented systems using methods like Booch's and HOOD.

The basis of the notation used by Wasserman *et al* is that of structured design used to describe modules and their inter-connections, e.g., calling structure and parameter passing. The notation represents a class with a rectangle and each method of the class is contained in a smaller rectangle superimposed on the border of the class symbol, representing the class's external interface; this arrangement looks very much like Booch's symbol for an ADA package. Hidden methods are allowed, shown completely enclosed within the class symbol. Attributes of a class are not explicitly mentioned or represented in the notation; the symbol for a private storage pool enclosed in a class is used to represent the attributes as a collected group hidden in the class (e.g., `stack data` in Figure 3.12); the notation for lexical inclusion of data modules in Structured Design is also used for representing attribute data. Calls to methods are shown as arrows going to the method boxes; parameters flowing into and out of methods use the same notation for parameter passing as in structure charts (e.g., item in Figure 3.12), with angled brackets denoting an instance of the class participating in the method (<stack> in Figure 3.12).

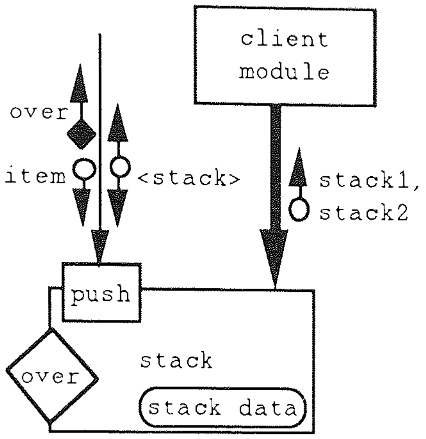


Figure 3.12 Example of OOSD notation.

A client/server relationship between two classes is shown as a thick arrow between the two parties involved in the using relationship. An output data flow appearing with such a client/server arrow indicates instantiation of the server class (e.g., stack1 and stack2 are instances of stack in the client module). OOSD facilitates the declaration of exception conditions represented as diamonds superimposed on the border of a class. The possibility of an exception condition being raised by a method (designated with a filled diamond) is shown with the parameter flow of the relevant method .

Inheritance between classes is represented by a dashed arrow from the subclass to its superclass (e.g., Figure 3.13 (a)). A subclass can add new methods and redefine inherited methods; the subclass may also add new attribute data, though again no mention is made of how individual attributes may be represented. Multiple inheritance is also allowed, with fully qualified names being used to disambiguate any conflicts. Generic classes are drawn with dashed borders with any formal generic parameters shown as parameter flows to the class symbol itself; generic parameters are represented originating from dashed circles. A generic class is used to generate a specific class by providing values for the parameters of the generic class. Instantiation of a generic class is denoted as a client/server relationship between the concrete and generic classes. A seniority structure is used to represent the use relationship between classes. There seems to be no mention of the parent-child relationship (or aggregation structure) between classes.

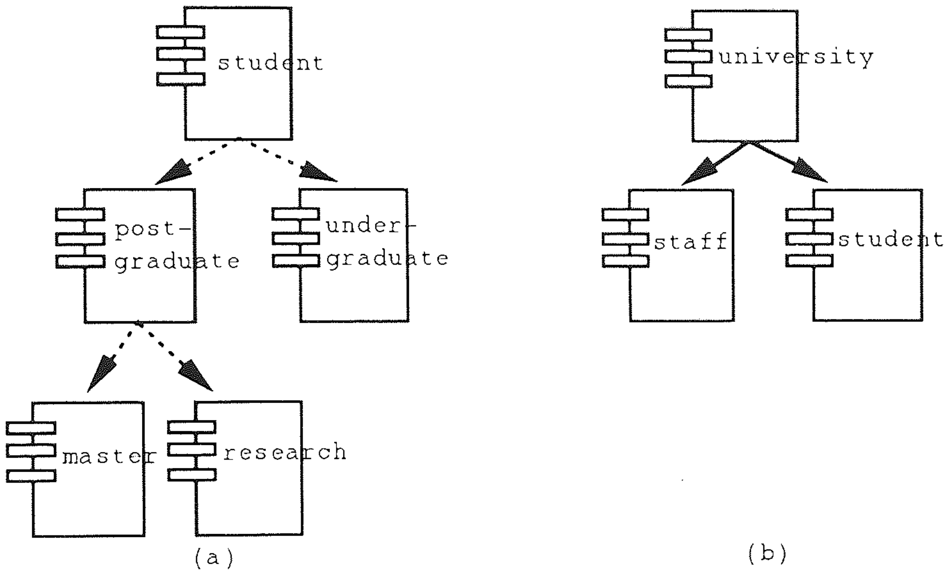


Figure 3.13 (a) Example class inheritance hierarchy in OOSD and (b) example seniority hierarchy in OOSD.

OOSD supports asynchronous processes in the form of monitors (similar to task in ADA) for handling concurrency management. Monitors are represented using Booch's symbol for an ADA task, where a parallelogram is used instead of the rectangle for a class. A monitor resembles a class except that its private data is shared by its various methods while each instance of a class has its own private state. Event-driven asynchronous activation (e.g., an interrupt) of a monitor operation is shown with a dashed line. The notation may have difficulty dealing with large numbers of methods, and there is insufficient provisions for representing complex data structures and attributes.

Comments on OOSD include: "OOSD is not so much a method as a notation to support object-oriented design methods in general" (Graham, 1991), and "the provision of specific method guidance is scant" (Walker, 1992). However, Wasserman *et al* (1990) note that it is often impossible to introduce or enforce strict design rules, and they have therefore "focused on a graphical notation without addressing the method by which a design should be derived".

HOOD

HOOD is an acronym for Hierarchical Object Oriented Design (HOOD Working Group, 1989, 1989a; Robinson, 1989). HOOD was a design method developed at the European Space Agency, aimed primarily at ADA development; as a result, "the HOOD methodology constrains itself and the user" to a subset of the design issues in object-oriented software development determined by the characteristics of ADA (Walker, 1992), restricting the general applicability of HOOD and its usefulness in a language-independent context. The design strategy is "globally top-down" and consists of a set of four basic design steps; these steps are repeated at each level of the top-down process. At each level, each of the objects identified so far, called the parent object, is decomposed into a set of component child objects which are viewed as providing the functionality of their parent. The decomposition process begins with the decomposition of the top-level parent object (root object) representing an abstract model of the system as a whole, and proceeds until it arrives at terminal objects which are designed in detail for direct implementation in code without further decomposition.

The first of the four basic design steps is the definition and analysis of the problem, involving selecting the relevant requirements for the design. Requirements of the system should already have been documented in a prior analysis phase. SADT (Ross, 1976; Ross, 1985) was chosen by the European Space Agency for its requirements analysis method. Here the designer states clearly the definition of the problem and the context of the system to be designed. The information gathered is analysed to make sure the problem is well understood.

The second step is to revise these requirements into an informal design strategy stated in English. This description should include references to real world objects and their actions, and is regarded as a baseline for further refinement in subsequent steps.

In the third step, major concepts of the solution strategy are extracted to try to formalise a solution. Objects and their operations (methods) are identified using a technique akin to Abbott's (1983) idea of using nouns and verbs respectively from the informal design strategy. Potential objects also include hardware devices to be localised, data to be stored and data to be transformed. The process is repeated as each object is decomposed. In decomposing the root object, child objects consist of real world problem domain objects, data, and data stores. At lower levels of decomposition, types of object include object states, data pools, records and output devices. Only objects at the right level of abstraction are selected at each level of decomposition, and all properties relating to operation execution (e.g., parallelism, synchronism, periodic execution) must be noted.

In the fourth step, operations are associated with objects, giving rise to an object operation table. HOOD diagrams can now be produced based on this table (Figure 3.14). Each diagram shows child objects and operations. The diagram should also contain *implemented-by* links between the parent object and its child objects, and use relationships, data flows and exceptions amongst child objects. An object symbol looks very much like a Booch ADA package symbol.

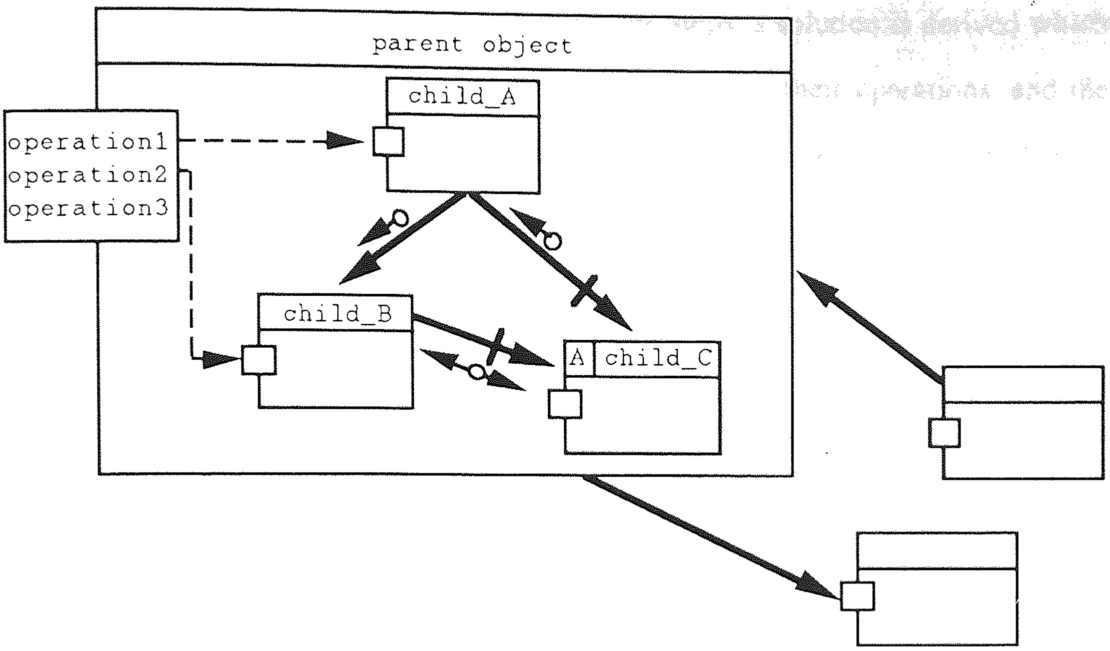


Figure 3.14 HOOD notation.

There are two type of objects in HOOD, passive and active (shown with an A in an object symbol e.g., child_C in Figure 3.14 is an active object). A passive object executes an operation immediately when control is passed to the object, while an active object's reaction to a stimulus may be delayed depending on its control structure. A use relationship is established between two objects when an object uses the operation of another object. Buhr's (1988) symbols for data flow have also been adapted (Robinson, 1989), mapping into ADA IN, OUT and IN OUT parameters. Cyclic use relationship among objects is prohibited to make testing more secure. A passive object may use only the operations of other passive objects, but this restriction does not apply to active objects. The use relationship is shown with a bold arrow from the using object to the used object (e.g., in Figure 3.14, child_A uses child_B and child_C, and child_B uses child_C). A parent object has an include relationship with its child objects, represented by the child objects' symbols included in the parent object symbol. A parent operation implemented by a child object operation is indicated by an implemented-by link shown as a dashed arrow from the former to the latter (e.g., Figure 3.14 shows that operation1 is implemented by an operation in child_A, and operation2 is implemented by an operation in child_B).

In the final step of HOOD's basic design steps, a solution is derived which describes each parent object in terms of its child objects, their operations, and the relationships among the objects. This information is produced as a formal *Object Description Skeleton* (ODS). The ODS includes an *Object Control Structure* (OBCS) for each active object written in ADA semantics in terms of the entry points and rendezvous semantics of each operation of the active object. Information in the ODS also includes a formal description of operations provided by the object, types, parameters, data flows and exceptions; the interface required by the object consisting of a list of used objects, types, operations and exceptions is also mentioned in the ODS. Each operation of the object will have an *Operation Control Structure* (OPCS) defining the operation's interface and logic using ADA pseudo-code.

There is no support for either inheritance or genericity. Generics is a mechanism often used in ADA programming, so its absence in HOOD, according to Hodgeson (1990), is a serious deficiency even for an object-based design method. Classes can be defined in HOOD. Instance objects of a class use operations defined in their class, but data types and data declarations must be defined explicitly in each instance, which seems not to exploit fully the abstraction potential of the class concept. Classes do not participate in an inheritance hierarchy, so reusability is not facilitated. Hodgeson (1990) observes that there is also insufficient separation between the definition of a class and its instantiation — the symbols for both class and instance, and their use, are identical in the HOOD notation. Hodgeson (1990) also identifies other deficiencies in the HOOD notation: data flows and exceptions (each exception is shown as a bar across a use relationship arrow) are both associated with a use relationship between two objects rather then defined for each operation; it is, therefore, impossible to indicate clearly which parameters and exceptions are related to a particular operation. Booch (1991) remarks that in HOOD, the role of data structures (attributes) is played down in favour of concentrating on functional abstraction.

Booch

Booch's (1991) object-oriented design method uses a notation comprising six different types of diagram. A *class diagram* is used to show the existence of different classes in the design of a system, and the various relationships amongst the classes, including inheritance, using, instantiation, and metaclass (see Figure 3.15 (a)). For inheritance, the notation allows for classes whose instances are not type-compatible with instances of its superclass (e.g., derived types in ADA). A using relationship between two classes may occur when the implementation of one class uses the resources of the other class, or when the interface of the former also depends on the latter — this is relevant in strongly-typed languages where the interface (messages) of the using class has to name the used class. The instantiation relationship is meaningful only for languages that support generic parameterised classes (e.g., ADA, Eiffel) where a class can be parameterised by other classes, objects and/or methods. Relative cardinality of classes can also be indicated for each relationship, specifying for an instance of a class the number of valid instances of another class participating in the relationship (see Figure 3.15(b)).

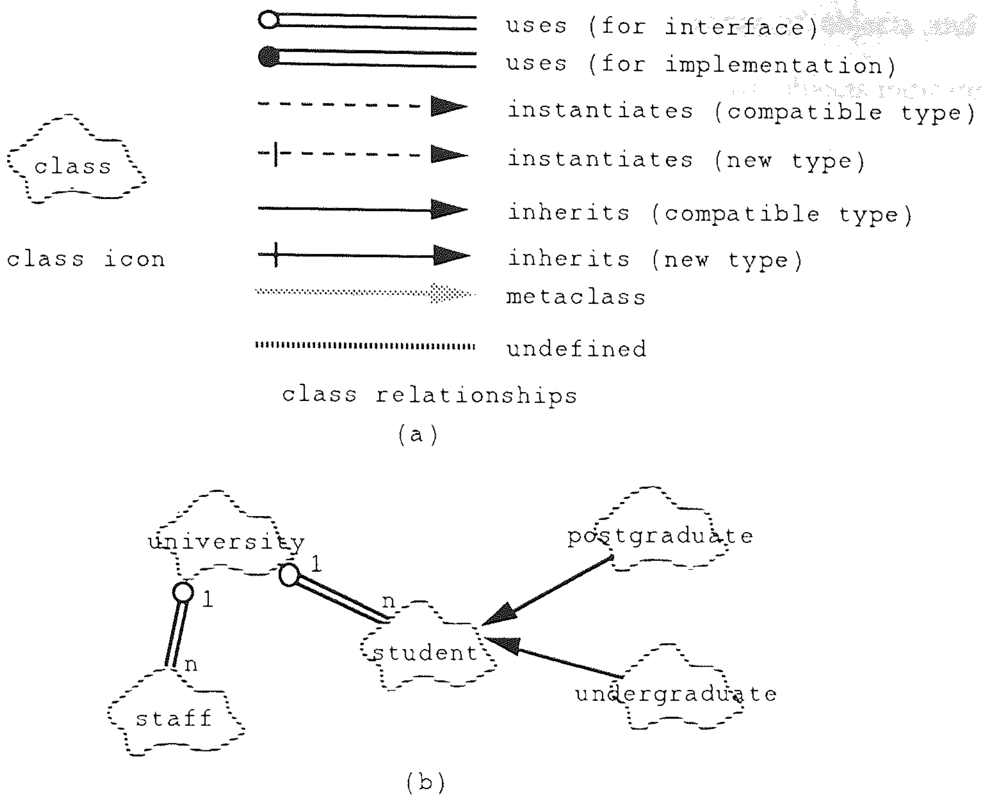


Figure 3.15 (a) Class diagram notation and (b) an example class diagram.

Booch's method copes with a complex system with a large number of classes by permitting logically related classes to be grouped into categories (cf. clusters and subjects). A class category in a class diagram refers to another class diagram containing the classes in the category. A top-level class diagram for a large system could therefore contain class categories (similar to the subject layer used by Coad & Yourdon (1990)) which in turn refer to lower-level class diagrams with further class categories and/or classes and their relationships. Each class is also described in more detail with a template form. The dynamic characteristics of each class are captured graphically using a state transition diagram. However, Graham (1991) notes that state transition diagrams are not appropriate when used with complex classes with very large numbers of states; e.g., an object with n number of boolean state variables will have 2^n different states.

An *object diagram* is used to show the existence of objects and their interactions (see Figure 3.16 (a)). A relationship between two objects indicates that the objects can send messages to each other; names of possible messages can be enumerated. There are three possible ways in which an object A may send a message to another object B (i.e., how A is visible to B): B is in the scope of A, B is sent as a parameter of one of A's methods, or B is a field (in the state) of A. An object's visibility to another object may also be shared with other objects via structural sharing, e.g., where B is referred to by other objects apart from A (Figure 3.16(a)). Message synchronisation semantics of interactions between objects is denoted using symbols adapted from Buhr's work (1988). Each object and message in an object diagram is described in detail using an object template and a message template respectively. The dynamic semantics of message passing is described using a timing diagram showing the flow of control amongst objects during the execution of an object method. Graham (1991) also points out that timing diagrams suffer from the problem of coping with a large numbers of objects and messages.

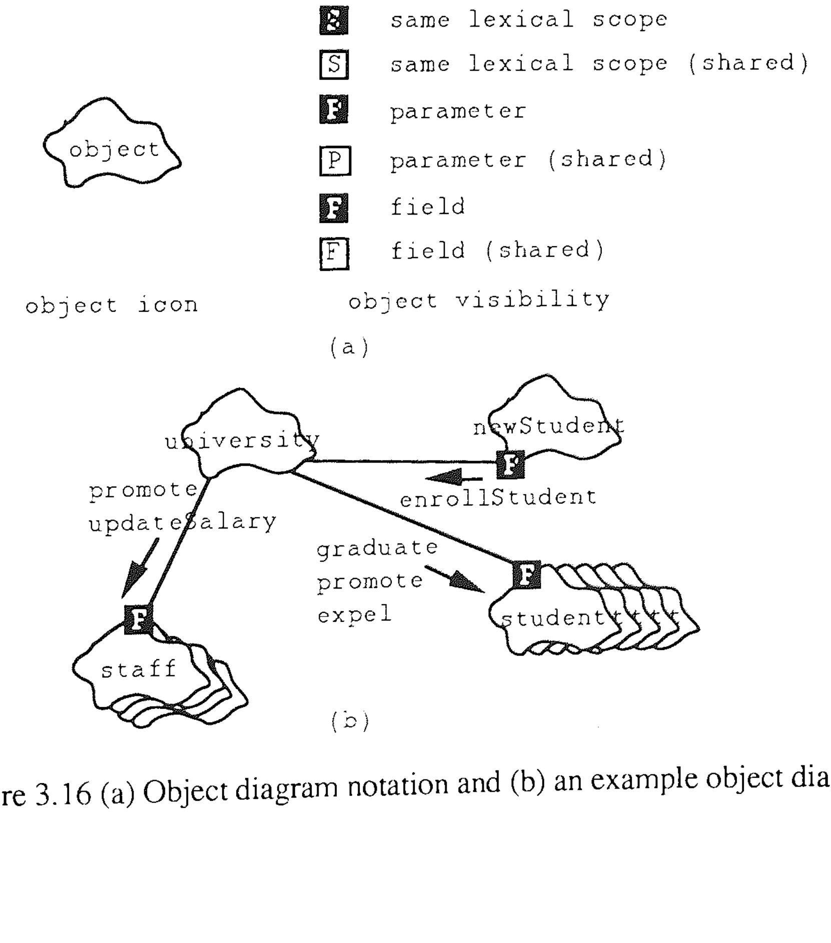


Figure 3.16 (a) Object diagram notation and (b) an example object diagram.

Class and object diagrams are used to document the logical design of a system. Booch's notation also uses module diagrams and process diagrams to support the physical design of the system. Module diagrams are used to show the allocation of classes and objects to modules, corresponding to separately compiled files in C++ and packages in ADA; connections in a module diagram represents compilation dependencies among modules. In some large systems, several programs are required to implement the design on a distributed system of computers. Process diagrams are used to visualise the allocation of processes to processors. Like block diagrams, a process diagram shows connections between processors and devices. Walker (1992) expresses concern that some of the diagrams in Booch's notation attempt to cover too much detail about a system. On the other hand, Walker feels that Booch's notation has also sacrificed detail or depth for breadth: "this can be useful at the early stages of the design process, but needs to be capable of being reworked to increase the level of detail". Detail is provided in part by textual templates, but Walker considers that there is insufficient correlation between the various templates and diagrams. Booch also does not explain how independent object and class diagrams are related to each other, in view of the fact that each object has to be an instance of a class.

Booch (1991) describes four main activities in his design method. The first step is to identify classes and objects, and then invent the *mechanism* (i.e., the message passing topology) which describes the "structure whereby objects work together to provide some behaviour that satisfies a requirement of the problem". Booch suggests that this step could be achieved by studying the problem's requirements and/or by discussing with domain experts to learn the vocabulary of the problem domain; tangible entities in the problem domain, and the roles they play, as well as events that may occur, all form candidate classes and objects of the design. Class and object diagrams are used here to show the various classes and objects and their relationships, and class and object templates can be started. The second step is to identify the semantics of the classes and objects, which means deciding on the interface (messages) of each class, and finding out which messages objects can send each other. This step could be iterative in that deciding upon the protocol of an object may require changes to decisions regarding the message protocol of another object. Booch suggests that writing a script for the life cycle of an object could be useful for identifying messages it can receive. The class and object templates started in the last step can be updated with information collected in this step. State transition diagrams and timing diagrams are used to document the dynamic semantics of classes and the mechanisms between class instances. New object diagrams might also be drafted to capture any new mechanisms invented in this step.

The third step is to identify the relationships amongst classes and amongst objects. This is largely seen as an extension of the activities of the previous step and involves two related activities, namely discovering patterns and making visibility decisions. Patterns among classes help in organising the class inheritance structure and patterns among cooperating objects can help in generalising mechanisms. Visibility among classes and objects refers to how classes and objects see one another. Classes see each other through relationships such as using, inheritance and instantiation, while objects can send messages to each other in the three ways mentioned above, taking into account the messages that can be sent between each pair of objects. Class and object templates are refined and completed to form the logical models of the design.

The fourth step is to implement the classes and objects. This involves looking at the internal representation of these entities, and also allocating the classes and objects to modules. At this point the design process may return to the first step, and the design process is applied to designing the inside view of existing classes and concentrating on lower-level abstractions. The result of this step includes refining the class structure of the system and completing the implementation part of each "important class template". Implementing classes and objects may lead to the discovery of new classes and objects, which may in turn result in refinement and improvement on the semantics of, and relationships amongst, existing classes and objects. The process of object-oriented design stops when "there are no new key abstractions [classes] or mechanisms, or when these classes and objects we have already discovered may be implemented by composing them from existing reusable software components" (Booch, 1991).

Pun & Winder

Pun & Winder (1989, 1989a) describe the framework for an object-oriented design method and its accompanying notation. The design process comprises three levels, *conceptual*, *system*, and *specification*. During the conceptual stage, application objects and interactions are identified from a requirements specification produced from the activities of a requirements analysis undertaken prior to the design process. Application objects are described as "objects which are understood by the client and end-users", in other words, application objects are problem domain objects. The method emphasises the importance of including the identification of user-interface objects at this stage with which users directly interact. The result of this stage is documented using an *object interaction diagram* showing the existence of the user interface and other domain objects ("user transparent objects") and their message connections.

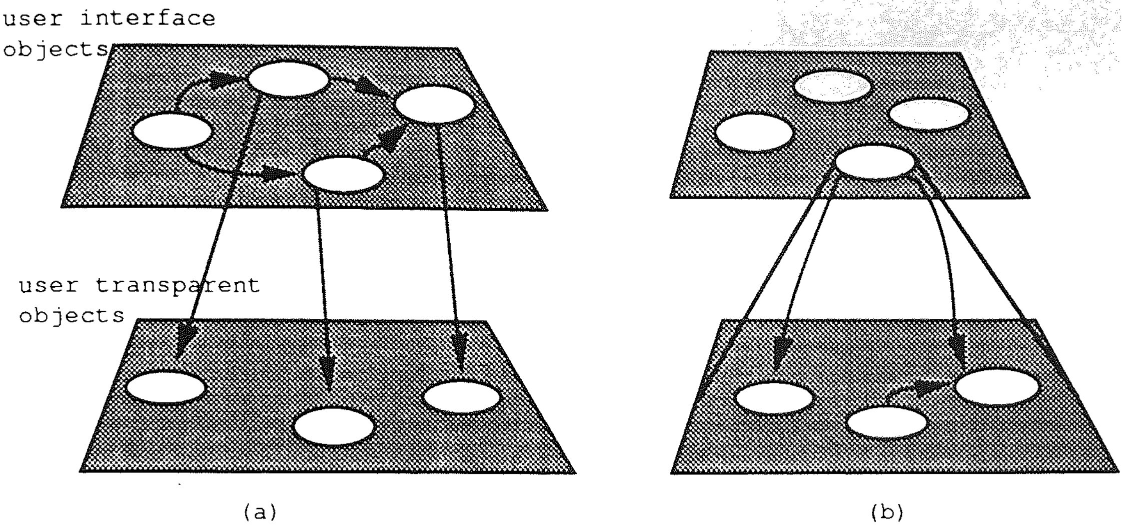


Figure 3.17 (a) An object interaction diagram and (b) a levelled object interaction diagram showing the contain relationship.

The system level is concerned with three kinds of relationships which can exist between objects: contain, use, and inherit. Looking at the contain and use relationships helps to identify implementation objects based on already identified application objects — implementation objects differ from application objects in that they are solution domain objects and must be implemented in the system e.g., an application object `user` identified in the conceptual level may not be implemented in the system. These additional objects are illustrated using levelled object interaction diagrams showing decompositions of objects, rather like levelled data flow diagrams (see Figure 3.17 (b)). The authors also mention an 'inheritance factorisation process' based on formal algebraic structure (Pun & Winder, 1989) which has been developed to help in the construction of a class inheritance hierarchy.

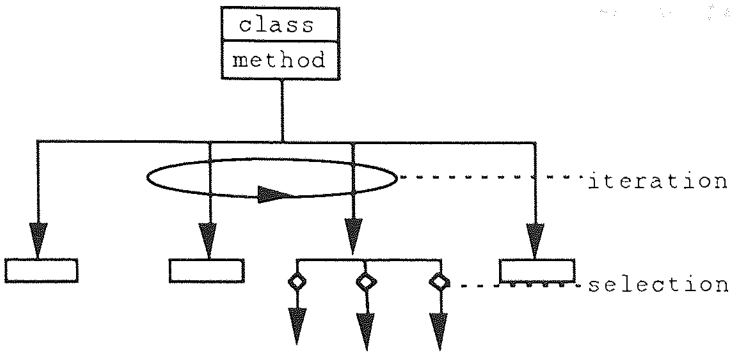


Figure 3.18 Notation for a class structure chart.

In the specification level, class structure charts are used to record information on each class including variables and messages, and the class's inheritance relationships with its superclasses. The method of each message in a class is also described in the class's class structure chart, showing the sequence of message passing and control information like iteration and selection (see Figure 3.18). In the notation for a class structure chart, a rectangle denotes a class and an arrow represents a message send.

Other Methods

Other methods for object-oriented design include work by Rumbaugh *et al* (1991) and Wirfs-Brock *et al* (1989, 1990). The Object Modeling Technique (OMT) described by Rumbaugh *et al* is aimed at the analysis, design and implementation of an object-oriented system. Influence of traditional methods is evident, including the use of entity relationship modeling (Chen, 1976) in the *object model* for representing associations among classes. These associations include aggregation, inheritance, *qualification*, etc.; a qualification is used to specify a restriction on a class in an association, e.g., in an aggregation relationship between a `directory` class and a `file` class, a qualification can be specified on the `file` class stating that each file instance must have a unique `fileName` attribute value in relation to a `directory` instance. State transition diagrams (based on Harel's state diagram notation (1987)) and modified data flow diagrams are used in the *dynamic model* and *functional model* respectively. State transition diagrams are used to describe the states and events of classes in the object model; actions associated with states and events refer to class operations. Data stores and *actors* (classes with attributes and operations) in data flow diagrams correspond to classes in the object model, and processes in low-level diagrams are described in terms of class operations.

Wirfs-Brock *et al* describe a technique called Responsibility Driven Design which models a system in terms of the *client/server model.* First, classes of objects are identified, and then actions which must be accomplished by the system are allocated to the classes to form the *responsibilities* of the classes (i.e., messages they can respond to). *Collaborations* with other classes (via message-sending) may be necessary to achieve class responsibilities in the client/server model. The set of messages that may be sent between a client and its server comprise a *contract* between the two classes. The class inheritance hierarchy in Wirfs-Brock *et al*'s method is structured based on a subclass inheriting only the responsibilities of its superclass rather than the superclass's structure (i.e., instance variables). The authors believe that concentrating on "object behavior before object structure" helps to "maximize encapsulation" in that structural information about an object does not become part of the interface to that object; thus, modifications to the object's structure is enabled without affecting the object's interactions with other objects. Wirfs-Brock *et al* use Beck & Cunningham's (1989) idea of recording the details of classes (including responsibilities and collaborations with other classes) on index cards; *collaboration graphs* are used to show graphically inheritance relationships and client/server relationships between classes. "The description of the dynamics of objects is atypical. There are no state-transition or data flow diagrams ... Instead, the behavior is formulated in terms of contracts, responsibilities, and messages" (de Champeaux & Faure, 1992).

Conclusion

This chapter has looked at the object-oriented software development life cycle models described by several authors. The essential feature they have in common shows that object-oriented development is an iterative and incremental process, and is also a combination of both top-down and bottom-up development. Several analysis and design methods proposed for object-oriented development have also been outlined.

Chapter 4 The Operational Approach and Object-oriented Software Development

4.1 Introduction

"Recently there have been complaints about the chronic problems of the conventional life cycle" (Zave, 1984). Balzer *et al* (1983) observe that maintenance on a system is traditionally performed on the system's source code (i.e., the implementation), and point out that the task of maintaining source code is made very difficult by optimisations in the code. The process of optimisation makes software harder to understand as programmers usually substitute (simple) abstractions with efficient but complex realisations. Optimisation also tends to increase dependencies among different parts of the system as programmers use knowledge from different parts of a program to facilitate optimisation, leading to scattering of related information; these dependencies are often implicit, thereby hindering maintenance further.

McCracken & Jackson (1981) note that the traditional life cycle ignores the fact that system requirements cannot usually be stated fully in advance, because often the user does not know completely what he requires in advance, or what the possible solutions are; the development process "changes the user's perceptions of what is possible, increases his insights into his own environment, and indeed often changes that environment itself". Swartout & Balzer (1982) consider the explicit separation of specification and implementation in the traditional model to be unrealistic, and argue that the specification and the implementation of a system are very much intertwined — both resource limitations encountered during implementation, and insights gained during actual implementation or while using the actual implemented system, may highlight any incompleteness in a specification. Implementation decisions, therefore, have the effect of modifying and refining the specification. Agresti (1986) notes other criticisms to the conventional life cycle model, which include the model's inadequate accommodation of prototyping, end-user development, and reusability.

4.2 The Operational Approach

New paradigms of system development have emerged in response to the problems inherent in the conventional approach and to challenge the conventional approach to software development; new ideas associated with these paradigms include prototyping, executable specification and program transformations, which do not fit into the conventional model. Executable specification and program transformation form an alternative strategy to software development known as the *operational approach.* Figure 4.1 shows the operational paradigm built around the preparation of an (executable) operational specification.

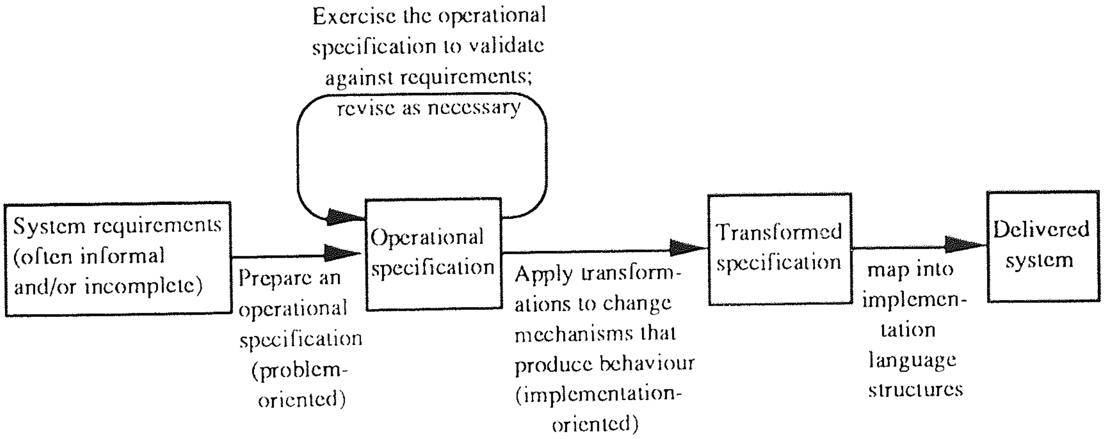

Exercise the operational specification to validate against requirements; revise as necessary
System requirements (often informal and/or incomplete)
Prepare an operational specification (problem-oriented)
Operational specification
Apply transformations to change mechanisms that produce behaviour (implementation-oriented)
Transformed specification
map into implementation language structures
Delivered system

Figure 4.1 The operational paradigm.

Operational Specifications

The first step in the operational approach involves the construction of an *operational specification.* "The operational specification is executable by a suitable interpreter" (Zave, 1984) to generate the behaviour of the specified system. An operational specification contains explicit descriptions of the intended behaviour of a system's operations in addition to the operations' interfaces (Liskov & Zilles, 1975); the descriptions of behaviour embody executable semantics and therefore enable the execution of the specification. The conventional life cycle model stresses the separation of external system behaviour — the "what" of requirements specification — from internal system structure — the "how" of design. Agresti (1986a) notes that separating software development activities on this basis introduces problems. Swartout & Balzer (1982) explain the inherent difficulty of extricating the "what" of a system from its "how" — discussing the "what" of a system requires design and implementation considerations to be addressed. Agresti (1986a) points out another problem associated with the rationale of separating behaviour from internal structure, in that when the design phase starts, the designer is left with a range of issues which impinge on design decisions. The designer needs to consider the following:

— problem-oriented issues of decomposing high-level functions into lower levels

— purely design issues such as information-hiding and abstraction

— implementation issues such as system performance constraints and the feasibility of implementing the design in the target hardware-software environment.

The operational approach acknowledges the intertwining of a system's functional behaviour and its internal structure. The guiding principle of this approach is not to overwhelm system designers by requiring them to deal with all these issues simultaneously. In the preparation of a specification, the operational approach partitions early development activities based on separating problem-oriented concerns from implementation-oriented issues. An operational specification specifies a system in terms of problem-oriented structures in some language or form which enables the specification to be executed. Zave (1984) explains that the internal structure of a system is explicit in an operational specification, while the system's external behaviour is implicit but can be revealed when the specification is executed, evaluated, or interpreted. "The operational specification resolves issues related to the behaviour of the system using terms that are meaningful in the user's problem domain. After the operational specification is prepared, the implementation-oriented issues can be addressed, without the confounding effects (experienced in the conventional model) of still wrestling with the functional processing of the system" (Agresti, 1986a). An operational specification uses structures that are independent of specific resource configurations or resource allocation strategies (and so can be implemented by a wide range of these) (Zave, 1984), which contrasts with a design of a system which is inevitably constrained by a specific run-time environment.

An executable operational specification serves as a prototype which produces the functional behaviour of the specified system, allowing the user to review the proposed system capabilities (Yeh, 1990). The way the specification generates behaviour may not relate to the actual run-time environment that will be used for the implementation of the system, and the behaviour is usually not efficient; this lack of efficiency is, however, not a main concern in preparing an operational specification. "The mechanisms (usages of the specification structures) in an operational specification are derived from the problem to be solved" (Zave, 1984). An operational specification is, therefore, structured to reflect the user's problem domain, hence enhancing comprehension by both the user and designer, and improving modifiability of the specification, regardless of any implementation characteristics. The key benefit of this prototyping capability is that validation of system behaviour by both the developer and the user can occur early in the development process. McCracken & Jackson (1981) and Yeh (1990) observe that prototyping provides an opportunity to understand and clarify the user's needs and his environment early. Agresti (1986a) points out that this early validation capability is difficult with traditional natural-language, static specifications which lack executable semantics. "It is not possible to see the behaviour of the system until parts of the specification are realised in code, which is normally late on in the development process" (Lewis, 1991).

Transformations

The transformation phase of the operational approach begins when a satisfactory specification has been produced after the end of the validation and revision loop (see Figure 4.1). The specification is subjected to a series of *transformations* which alter or augment the behaviour-producing mechanisms of the specification while preserving the external behaviour of the specification. The resulting transformed specification specifies the same system in terms of implementation-oriented structures that will eventually be mapped into the implementation language. Zave (1984) identifies two main types of transformation. One type of transformation changes the mechanisms of the operational specification to balance performance and implementation resources. Other transformations are needed to manipulate the structures of the specification so that they can be mapped straightforwardly and efficiently onto a particular configuration of implementation resources (such as processes, memory, communication channels, etc.). This second type of transformation on structures may introduce explicit representations of implementation resources or resources allocation mechanisms that were not present in the original specification.

The process of transforming an operational specification could potentially be automated, addressing "the labor intensiveness of software development by using specialized computer software to transform successive versions of the developing system mechanically" (Agresti, 1986a). In addition, a knowledge-based software "assistant" (Balzer, Cheatham & Green, 1983) could be used to guide the developer during the transformation process in selecting the appropriate transformations (e.g., identifying candidate transformations, choosing data structures, etc.), ensuring correctness at each step. Automation of the transformation process offers the capability of performing maintenance on a system's specification rather than on its implementation (source code). Changes to a system's functions may be achieved by revising its specification, which is then re-implemented with computer assistance. Yeh (1990) observes that this transformation approach reduces "the opportunity for new errors to be introduced during modification and ensures error minimization as the system evolves in response to changing user circumstances". Balzer *et al* (1983) state that "by maintaining the specification directly, we drastically simplify the maintenance problem" as the specification is "the form that is closest to the user's conceptual model, least complex, and most localized ... before optimization decisions have been integrated, so modifications are almost always simple, if not trivial. We are constantly reminded of this insight by end users and managers who understand systems only at this (unoptimized) specification level; they have no trouble integrating new or revised capabilities in their mental models". Balzer *et al* also envisage that, with suitable specification languages, users should be able to develop and maintain specifications themselves, improving the interface between users and implementors. With maintenance of specifications, the "goal of reusable software can be attained"; specifications are placed in libraries and when a module is required, a suitable specification is reused by modifying it appropriately and then re-implementing it with automated transformations.

4.3 Overview of Some Operational Methods

JSD

JSD specifies a system in terms of a network of long-running sequential *processes* which communicate by writing and reading from buffers called *datastreams*, and by inspecting each other's internal states, known as *state vectors*. Figure 4.2 shows an example JSD *network* specification which describes the abstract architecture of the system. Processes are represented by rectangles, and datastreams and state vector inspections are shown as circles and diamonds respectively. Beside *model processes* that model real-world entities, other processes known as *function processes* are added to a specification which realise the functional requirements of a system. Function processes are used to handle data collection (to ensure error-free input), to extract information from model processes and produce output, and to perform extra processing and then feed results back to model processes.

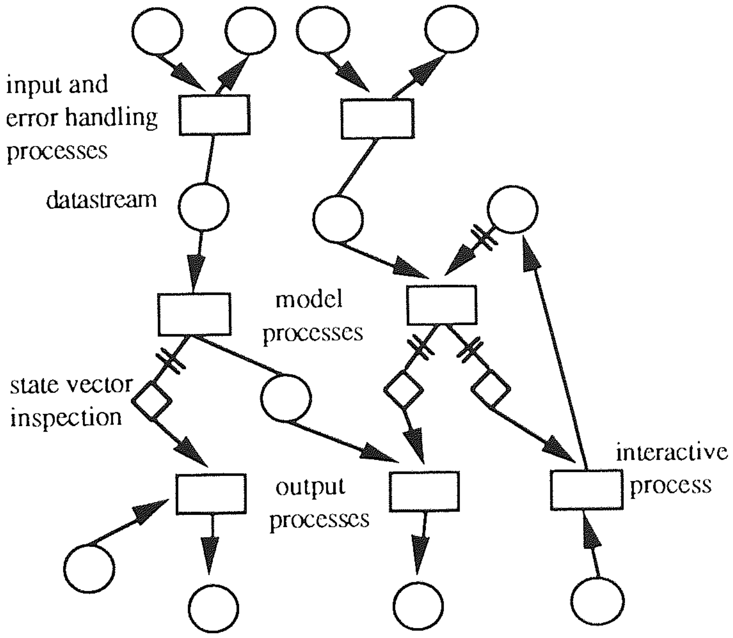


Figure 4.2 An example JSD specification.

Each model process describes the time-ordering of events (or *actions*) suffered or performed by a real-world entity. Representing real-world entities as time-ordered event structures is known as *entity life modelling* (Cameron, 1988; Sanden, 1989). For example, a student registers as a student of a university, passes the required examinations, before finally graduating; a student cannot violate this order of events by graduating before registering or passing all the examinations. Figure 4.3 is an example process diagram showing the time-ordering of actions for a Book entity of a library. Actions are represented by leaf rectangles, and iterations and selections are denoted by the symbols '*' and 'o' respectively. Actions are atomic in that they cannot be decomposed into sub-actions; moreover, actions are considered to happen instantaneously. Each action has associated attributes which describe its characteristics; for example, the action `Lend` in Figure 4.3 would have attributes such as `book-id`, `ISBN`, `borrower-id`, `date`, etc. An entity's attributes may be derived from the attributes of the entity's actions, e.g., a `Book` entity would also have attributes such as `book-id` and `ISBN`.

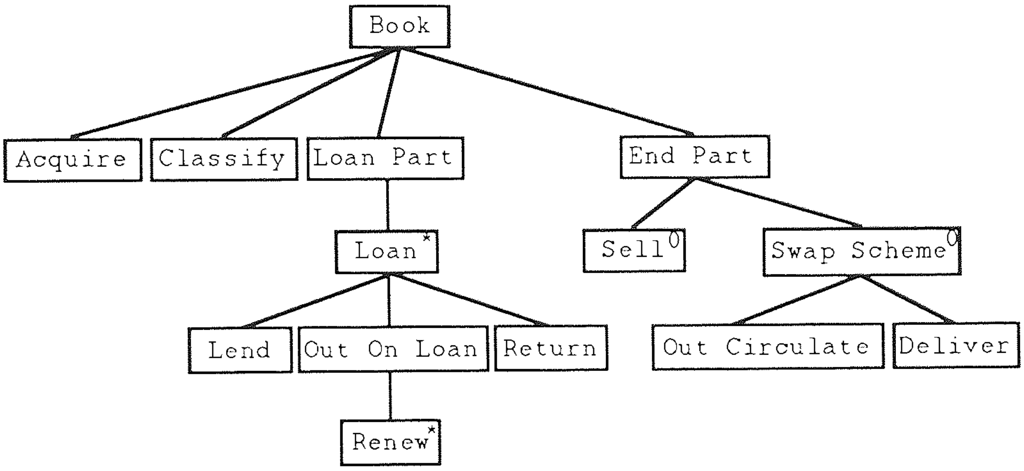
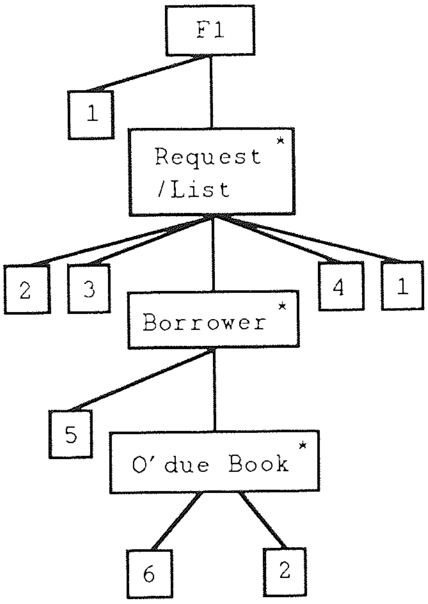


Figure 4.3 A process diagram for a `Book` entity.

For each similar entity that exists in the real world, a separate model process is regarded as existing which models the entity's life history. A model process type and its actual instances may be considered as reflecting the class-instance relationship present in an object-oriented environment. Each model process instance writes its own datastream and its state can be inspected by a state inspection. Process multiplicity is denoted by the symbol ═ in a JSD specification diagram (e.g., Figure 4.2). Entities belonging to a particular process type have common attributes which form the individual state vectors of corresponding model process instances. The state vector of a model process instance can be manipulated only by the actions of the process. Identities of process instances form a part of their state vectors; however, the use of identity as part of the state of an entity has been noted as undesirable in the object-oriented sense (see Section 2.2).

Using a process diagram, the internal structure of a function process in a specification is also described in terms of the ordering of actions the process has to perform. Actions of both model and function processes are defined by attaching primitive operations — such as reading and writing of datastreams, assignments to update process attributes, and state inspections — to the actions. For example, in a library system specification, the process diagram of a function process, `F1`, which periodically lists the overdue books grouped by borrower, may be annotated with primitive operations as shown in Figure 4.4 (Cameron, 1986).



1. read next REQUEST
2. read next BOOK SV (overdue books only)
3. write HIST-HDR
4. write LIST-TRLR
5. write BORROWER-HDR
6. write BOOK-LINE

Figure 4.4 Example JSD process diagram annotated with primitive operations.

Lewis (1991) refers to the datastream communication mechanism in JSD as 'message sending'. This message sending mechanism is asynchronous; a process can write a message to a datastream without being blocked but will get blocked when reading from an empty datastream. This message sending mechanism in JSD differs from the message sending mechanism in the object-oriented paradigm — Wolczko (1988) notes that the object-oriented messaging mechanism is regarded as synchronous. Lewis (1991) also observes that "asynchronous communication does not generally exist in the object-oriented paradigm", as an object in an object-oriented environment is blocked on sending a message to wait for the response of the message receiver.

A state vector inspection between two processes permits one process to inspect the state of the other without interruption to the inspected process, and the inspecting process is never blocked in its attempt. However, "the results of inspections [must] correspond only to particular coherent states" of an inspected process, i.e., the state of the process "just before the execution of a read operation" (Cameron, 1986). This restriction means that an inspection must be delayed when the inspected process is updating its state while executing an action. No mechanism equivalent to state vector inspection exists in the object-oriented paradigm. Graham (1991) points out that state vector inspection directly violates the principle of information hiding. Encapsulation in an object-oriented environment ensures that access to the values in an object's state may be achieved only by the usual way of sending the object appropriate messages provided in the object's protocol which return values of its instance variables.

Other existing operational methods include Gist (Balzer, Goldman & Wile, 1982; Cohen, Swartout & Balzer, 1982; Feather, 1982), Paisley (Zave, 1982; 1984) and Me-Too (Henderson, 1986). A brief overview of the specification technique of each is provided next.

Gist

Gist, like JSD, aims to model the problem-domain explicitly in its specifications. A Gist specification is expressed in terms of descriptions of behaviours which "correspond to the observable activities in the application domain". Behaviours are specified using ‘stimulus-response rules’ called demons. A specification also consists of a state comprising a set of abstract objects; activities in the problem-domain trigger the relevant rules which map the current state of the specification into a new state by modifying the values of its state objects. For example, the demons and object types in a package routing example (Balzer *et al*, 1982) may be specified in the Gist language as shown in Figure 4.5. In the package routing system, “at random times, a new package appears at a particular location called the source assigned to be routed to an arbitrary destination bin. Packages located at the source are moved to their destinations”.

```
demon CREATE_PACKAGE()
response
        create package.new||package.new:DESTINATION=a bin and
                package.new:LOCATED_AT=the source

demon MOVE_PACKAGE(package)
trigger package:LOCATED_AT=the source
response
        update LOCATED_AT of package to package:DESTINATION

type package(located_at|location)
type location supertype of <source;bin>
```

Figure 4.5 A Gist specification example.

Paisley

Paisley is similar to JSD in using processes to model the real-world in its specifications. A process simulates some part of the problem-domain and executes indefinitely. Each process is specified by a ‘successor function’ using a functional language based on side-effect-free “expressions formed from constants, formal parameters, functions and functional operators”. The process’s successor function is used to generate the process’s new state from its current state. For example, the successor function of a producer-consumer buffer, `next-buffer`, may be specified in Paisley as shown in Figure 4.6, where the buffer’s state is the current contents of the buffer.

```
next-buffer:BUFFER->BUFFER;
next-buffer[b]=give-to-consumer[get-from-producer[b]];

get-from-producer:BUFFER->BUFFER;
get-from-producer[b]=
/full[b]:b,
'true':put-on-tail[(b,xr-prqd['null'])]
/;

give-to-consumer:BUFFER->BUFFER;
give-to-consumer[b]=
/empty[b]:b,
'true':put-on-head[(xr-cons[first[b],rest[b])]
/;
```

Figure 4.6 A Paisley specification example.

Me-too

Me-Too uses a combination of the functional language Miranda (Turner, 1986) and the formal language VDM (Jones, 1990) to formally express the abstract data objects and operations of a system in terms of abstract data types and recursion equations. A system is modelled by a set of state variables of abstract data types which "can be assigned values of abstract objects" (Henderson, 1986). Functions defined for abstract data types are used to construct new abstract objects to be assigned to system variables. To illustrate the Me-too specification language, Henderson (1986) uses the example of a database which can store a collection of notes (English phrases) and which can be searched for recorded notes containing occurrences of a selected phrase. Figure 4.7 shows the abstract data types required in the example and some functions of the type Db.

```
Note=seq(Word)
Db=set(Note)

emptyDb:->Db
addNote:Db×Note->Db
delNote:Db×Note->Db

emptyDb()≡empty
addNote(db,n)≡db»{n}
delNote(db,n)≡db-{n}
```

Figure 4.7 Example of abstract data types and functions in Me-too.

Operations provided by the system may be specified as shown in Figure 4.8.

```
mentions:NoteXWord->Boolean
mentions(n,w)≡or/<w=w'|w'<-n>

mentions-all:NoteXNote->Boolean
mentions-all(n,n')≡and/<mentions(n,n')|w'<-n'>

all-mentions:DbXNote->Db
all-mentions(db,n)≡{n'|n'<-db;mentions-all(n',n)}
```

Figure 4.8 Example system operations in Me-too.

Specification Executability

As stated earlier (see *Operational Specifications* in Section 4.2), explicit descriptions of a system's operations provide the executable semantics in an operational specification, enabling its execution. In Paisley and Me-too, semantics of operations are specified in a functional language. Functional languages are interpretable and therefore support the executability property of operational specifications. For example, a process in Paisley may be executed by repeatedly replacing its current state by a successor state obtained by evaluating the expression representing its successor function. Me-too specifications may be exercised using a "read-eval-print loop ... typical of interactive programming systems, especially Lisp systems". In the Me-too example above, once the system is initialised (`db:=emptyDb()`), the sytem may then be executed by issuing commands such as:

```
db:=addNote(db,"result of a function")
all-mentions(db,"value parameter")
```

On the other hand, Gist and JSD both use an imperative style in specifying their operations. Gist and JSD specifications are only executable in principle. "Gist specifications can be evaluated to yield behaviour (a sequence of states) given an initial state" (Balzer *et al*, 1982). When a demon's trigger (a particular system state) becomes true, the demon is activated and modifies the system state. The new state may then trigger another demon and the cycle is repeated. However, Balzer *et al* admit that the execution of Gist specifications "is not possible directly because the evaluation of Gist specifications in (*sic*) intolerably slow". Transformations to convert Gist specifications into a suitable execution form (Feather, 1982) and the use of symbolic evaluation techniques (Cohen *et al*, 1982) are two possible ways "for allowing Gist specifications to be used as prototypes" (Balzer *et al*, 1982).

In a JSD specification, each long-running model process executes constantly, reading inputs and carrying out actions to "coordinate itself with the reality" of its corresponding real-world entity. Each model process takes as long to execute as its corresponding entity. The execution of a JSD specification is dependent upon finding a machine/operating system that will handle the execution of a large number of instances of concurrent long-running processes. Cameron (1986) states that to execute a JSD specification, "we often have to combine and package the specification processes into a more familiar arrangement of "short-running" jobs and transaction-handling modules". A basic technique is to convert each process into a subroutine by inserting a suspend-and-resume mechanism at its read statements and by passing input records as parameters of the call (Jackson, 1983). Every time a subroutine is called, it executes part of the long-running program (Cameron, 1986).

4.4 Adopting an Operational Approach in Object-oriented Development

Given the claimed advantages of the operational approach to system development, object-oriented development could benefit from these advantages by incorporating the operational characteristic of creating executable specifications as part of the object-oriented development process. One possible route to this aim is to use elements of an existing operational method in an object-oriented design context to construct executable specifications of systems which are then implemented in object-oriented languages.

Research has been carried out by Lewis (1991) to explore the feasibility of using JSD in this respect. However, in comparing the two paradigms, Birchenough & Cameron (1989) state that "JSD is explicitly object-oriented only during the modelling phase". Lewis (1991) adds that although "there are superficial connections which have been identified at the specification phase of JSD ... the differences between the two domains are such that JSD cannot really be categorised as object oriented (*sic*)". Due to the differences between the abstractions used in JSD specifications and the concepts of the object-oriented paradigm, in using JSD as a requirements specification technique with object-oriented development, a transformation phase between the specification of a system and the system's implementation becomes mandatory to map between the two paradigms. Lewis (1991) describes some transformation strategies which may be used to map JSD specifications into Smalltalk-80 implementations, but also explains the attendant problems and constraints associated with applying these strategies.

Important points which make the use of JSD specifications unsuitable for object-oriented development include the absence of support for inheritance and the containing (or association) relationship between objects (see Section 2.2). Gist, on the other hand, does provide a supertype-subtype relationship between data types which may be used to represent inheritance. The containing relationship may also be expressed in Gist as shown in the example in Figure 4.5 (e.g., between the types `package` and `location`). However, it has no mechanism in its demon definition, similar to the write primitive of JSD, which may be used to represent a message sending relationship between objects. Also, all demons in Gist share a database of state variables, making it difficult to represent encapsulation of state and processing. The using relationship amongst objects (see Section 2.2) cannot be represented in Gist (cf network diagrams of JSD), and so animation of specification execution is not facilitated (see *Executable Semantics and Execution Animation* overleaf).

There is also no support for inheritance in Paisley or Me-too. In Paisley, the only type of data object is the list or sequence, and substructure within these structures "cannot be explicitly acknowledged" (Zave, 1982). This makes it difficult to represent containing relationships between objects. Side-effect-free functions in Paisley cannot represent the concept of a method which must have update privilege to an object's state and be able to send messages to the state objects. Me-too, however, borrows VDM's syntax for the description of record or tuple structures, which may be used to represent the containing relationship between objects. Standard constructors for, and selectors on, tuples are also provided, which facilitates the definition of methods. The using relationship cannot be represented explicitly in either Paisley or Me-too, and thus, as with Gist, animation of specification execution is not facilitated.

Instead of using an existing operational specification technique, another approach would be to use an existing object-oriented method to create operational specifications which ensures that important features of the object-oriented paradigm are addressed. The subsequent implementation of the specifications in an object-oriented language would be facilitated by avoiding the need for transformations required in bridging the gap between two different paradigms. First, however, it is necessary to determine the dynamic properties and executable semantics present in an object-oriented system. These dynamic properties and executable semantics could then be used to assess the extent to which the notation in each object-oriented method described in Chapter 3 provides support for building operational specifications of object-oriented systems.

Operations of an object-oriented system are provided by object methods. A medium is needed in the specification for describing explicitly the internal details of methods to provide the executable semantics required to enable the specification's execution (see *Specification Executability* in Section 4.3). The description of a method must express the messages sent to other objects within the method to achieve the method's task.

Animating the execution of a specification of an object-oriented system requires the dynamic behaviour of the system to be represented explicitly in the specification. The dynamic behaviour of an object-oriented system may be viewed as relating to two separate aspects: the system's functions and objects in the system.

An object-oriented system consists of a set of objects interacting dynamically to provide the required functionality of the system. To address the function aspect of the dynamic behaviour of a system, a specification must contain a model which expresses the dynamic message interactions of objects in the system. A particular characteristic of the operational approach is its emphasis on constructing a specification of a system which represents "an operating model of the system functioning in its environment", showing the system interacting with its environment (Zave, 1982). The specification model of an object-oriented system must therefore show the effect of the environment on the system in terms of external sources of messages to objects in the system.

Objects in the system may be seen as active entities receiving messages and responding by manifesting their own behaviour while contributing to the overall functionality of the system. The dynamic "lifestyle" of each individual object in terms of the messages the object receives in its life-time forms the second (or life-history) aspect of the dynamic behaviour of a system and should be recorded in the specification of the system. This aspect of a system's dynamic behaviour is similar to entity life modelling mentioned earlier (see Section 4.3) used for modelling the dynamic behaviour of real-world entities. The function and life-history aspects of an object-oriented system's dynamic behaviour need to be captured in detail in order to facilitate the animation of the dynamic message sending and receiving activities of objects during an object-oriented system's execution.

Analysis of Some Existing Object-oriented Methods

Most of the object-oriented methods described in Chapter 3 use state transition diagrams and data flow diagrams to represent dynamic information about an object-oriented system. State transition diagrams (STDs) are used to represent the dynamic behaviour of individual objects over time in the methods OOSA (Shlaer & Mellor, 1988), OMT (Rumbaugh et al, 1991), and that of Booch (1991). But as Graham (1991) points out, the use of STDs is not practical where a very complex object is concerned which may have a large number of states. Moreover, the state of an object may comprise other complex objects; the precise definition of each state that would be required in order for the STD to be executable is therefore difficult to construct. In this respect, STDs are not effective for modelling the dynamic behaviour of objects. Booch and OOA (Coad & Yourdon, 1990) suggest using STDs for representing method details. Used in this context, an STD is only able to show what `self` or `this` messages (Smalltalk and C++ terminology respectively) an object needs to send itself to execute the method, but not what messages need to be sent to other objects in collaboration. Moreover, it is difficult to represent any sequencing of message interactions which Bailin (1989) considers "a necessary concept in specifying operational scenarios of a system".

Data flow diagrams (DFDs) are used in OOSA, OOA and OMT for different purposes. Coad & Yourdon (1990) use DFDs in OOA for defining services (i.e., methods). A DFD can implicitly represent message interactions with other objects in a method when interpreted using the strategy proposed by Alabiso (1988) for converting DFDs into object-oriented designs. It is difficult, however, to represent any sequencing of these interactions. In OOSA, DFDs are used to denote data flow between states in STDs, each state having associated actions which an object must carry out on reaching the state. In this context, DFDs cannot really represent dynamic object behaviour or details of methods. Coad & Yourdon (1990) make the same observation and state that OOSA does not really provide any mechanism for expressing the concept of method. Another problem with OOSA's use of DFDs involves identifying which data flow relates to which action for a given state, as each state can have several actions attached. OMT uses DFDs to model the system structure in terms of data flow amongst classes. Monarchi & Puhr (1992), however, express concern about the difficulty of recognising "how an object, behavior or attribute ... relates to a data flow or data flow process". A DFD is not, therefore, very useful for representing the system function aspect of the dynamic behaviour of a system. Alabiso (1988) provides *function design charts* in his notation to express the "make up of Methods (*sic*)" in terms of message interactions with other objects. However, the notation does not provide sequencing constructs for representing flow of control.

Explicit representation of method details is provided only in Booch's method and the approach of Pun & Winder (1989). Timing diagrams in Booch's method depicts the flow of control amongst objects during the execution of a method. Pun & Winder's notation includes class specification charts for defining each method in terms of the sequencing of message sends to attribute objects. Also, object diagrams and object interaction diagrams respectively in Booch's and Pun & Winder's notation explicitly model the system function aspect of dynamic behaviour in terms of message interactions amongst objects in a system.

In summary, no single method provides representations for the executable semantics, and the function and life-history aspects of the dynamic behaviour in an object-oriented system. Most notably the life-history aspect of dynamic behaviour described previously cannot be modelled adequately, and hence animation of specification execution cannot be supported fully by using the notation of any existing method. A new operational specification notation is needed which addresses equally the executable semantics and dynamic behaviour in an object-oriented system. The new notation could also incorporate useful features of some of the existing methods.

4.5 Conclusion

This chapter has outlined the operational paradigm of system development. The operational approach constructs executable specifications which can serve as system prototypes, thereby offering the potential to overcome some of the problems associated with the traditional approach to system development. It is proposed that object-oriented development could also benefit from an operational approach by using executable specifications. Different approaches for how this might be achieved have been considered. It is suggested that a new operational specification notation is required for specifying object-oriented systems which embodies executable semantics to enable specification execution, and at the same time represents dynamic behaviour explicitly so that animation of specification execution is facilitated. The next chapter discusses how behaviour and executable semantics may be specified using the new notation, and describes a method for developing operational object-oriented specifications using the notation. Chapter 6 introduces a tool which has been implemented to support the method and which can animate the behaviour expressed in the specifications constructed.

Chapter 5 Operational Object-oriented Development

5.1 Representation of Dynamic Behaviour and Executable Semantics

Dynamic Behaviour

Atkins & Brown (1991) state that objects are said to exhibit behaviour because an object can receive messages that invoke "operations which selectively reveal or manipulate the object's state". This object behaviour gives rise to two important aspects of dynamic behaviour in object-oriented systems — system function and object life history (see Section 4.4) — which need to be explicitly represented in a specification to facilitate animation of the specification's execution.

An object-oriented system's behaviour stems from the message interactions of objects in the system. The importance of inter-object behaviour is recognised by other authors who refer to this behaviour in terms of *collaborations* (Wirfs-Brock & Johnson, 1990), *responsibilities* (Wirfs-Brock & Wilkerson, 1989), *contracts* (Helm *et al*, 1990), and *mechanisms* (Booch, 1991); the authors emphasise the description of the "behavioral dependencies" (Helm *et al*, 1990) amongst objects in an object-oriented system, where a client object's behaviour depends on a server object's services (i.e., messages) (Wirfs-Brock & Wilkerson, 1989). The new notation being proposed should, therefore, be able to represent behavioural dependencies and client/server relationships amongst objects. Apart from the object-oriented methods described in Chapter 3, the notation of other development methods will now be considered as to their appropriateness for representing this system behaviour. The 'activity model' in SADT (Ross, 1976) and the 'dynamic diagram' in CORE (Mullery, 1979) resemble data flow diagrams in that they are used to show flow of data amongst activities in a system. Earlier criticism (in Section 4.4) against using data flow diagrams for representing behaviour of object-oriented systems applies to these notations also. More importantly, a data-flow notation cannot represent explicitly the behaviour dependencies and client/server relationships of objects. Petri nets (Peterson, 1981) used for real-time system development describe a system in terms of a set of transitions representing processing steps or events, and a set of places representing system states between the transitions. It has been noted in Section 4.4 that state transition diagrams are not practical for describing the behaviour of complex objects which may have large number of states (Graham, 1991). A similar problem could arise in using a petri net to describe an object-oriented system because the net would have to include places to represent all the possible states of all objects, and any combinations of these, in the system. Moreover, behavioural dependencies and client/server relationships amongst objects are not directly represented and will have to be inferred from the places and transitions in a petri net. Petri nets are therefore not ideal for modelling the system behaviour of an object-oriented system.

The system function aspect of an object-oriented system's dynamic behaviour may be best represented in the new notation, using a network model similar to the object diagram or the object interaction diagram of Booch's and Pun & Winder's notation respectively. In addition, given the operational emphasis on the specification reflecting the system functioning in its environment, the model must be enhanced to include the system's interactions with its environment in terms of external sources of message to objects in the model.

The life history aspect of dynamic behaviour relates to individual objects as independent entities receiving messages over time. Object-oriented methods described in Chapter 3 which address dynamic behaviour of objects use state transition diagrams to represent this object behaviour. The limitations of describing object life history in terms of state transitions have been discussed in Section 4.4. Booch (1991) states that "the existence of state within an object means that the order in which operations are invoked is important". The life history of an object may alternatively, therefore, be modelled in terms of the temporal ordering of messages the object receives, similar to entity life modelling in JSD and SSADM (Ashworth & Goodland, 1990).

Purchase & Winder (1990) consider the ordering of messages received by an object a vital issue in implementation; they describe a specification mechanism called *Message Pattern Specification* (MPS) in the parallel object-oriented programming language Solve (Roberts *et al*, 1988) for expressing "legal patterns of run-time behaviour" for objects as part of a system's implementation. The specification of a system, therefore, also needs to address this view of dynamic behaviour of an object, i.e., as the sequencing of messages an object receives in its life-time. The entity life history diagram in SSADM and the process structure diagram in JSD could be adapted for this purpose in the proposed notation, where events and actions of entities and processes are interpreted as messages of objects.

Executable Semantics

Definitions of methods in an object-oriented system provide the executable semantics required in an operational specification to enable the specification's execution. Timing diagrams used by Booch (1991) and class structure charts in Pun & Winder's notation (1989) for representing method definitions concentrate only on showing the flow of control in a method as messages are sent to other objects. However, "the action [of a method] following the message send is a function of the present object state, the message selector, and its arguments" (Purchase & Winder, 1991). Representation of method detail of an object in the new notation must therefore provide for all of the following elements:

a. the method's message selector and arguments

b. the effects on/of the state of the object

c. the messages that need to be sent to other objects to help achieve the goal of the method

d. the sequencing of (b) and (c) in the method.

SREM (Alford, 1985) — a method for developing distributed real-time systems — use *R-nets* to specify 'stimulus-response' processing in a system. "Each R-net specifies the transformation of a single input message plus current state into some number of output messages plus an updated state" (Alford, 1985). It appears possible to use an R-net to express the detail of an object method. However, because SREM has been developed for the procedural paradigm, modifications to the R-net notation would be required, e.g., to convert ALPHAs which are procedure-calls in an R-net into message sends and their receiver objects. It has been decided, therefore, that new constructs for method definition will be built into the new notation instead of using modified R-nets. In addition, the syntax and semantics of Smalltalk-80 will be used to provide the formal basis required for the design of the method definition constructs so that a new grammar for the constructs will not have to be defined.

The new notation and the development process for using it to construct operational specifications of object-oriented systems are described in the rest of the chapter.

5.2 The Development Process

The activities of developing an operational specification of an object-oriented system may be divided into four basic steps:

1) identify the objects

2) establish the interactions amongst the objects

3) determine the semantics of the objects

4) implement the objects.

These steps look familiar as they have been described in other object-oriented development methods in various forms; thus, the steps to be described are not revolutionary. However, the aim of the development method being proposed here is to construct an operational specification of an object-oriented system; the operational specification can then be executed and the execution animated. The following sections describe the activities in each step of the development process, wherein the system's executable semantics and dynamic behaviour (discussed in the Section 4.4) are expressed in terms of a new graphical notation.

5.3 Object Identification

The first step of the process to develop an operational specification of an object-oriented system is to identify the main objects in the system. These objects represent the key abstractions in the problem domain: "they give boundaries to our problem; they highlight the things that are in the system and therefore relevant to our design, and suppress the things that are outside the system and therefore superfluous" (Booch, 1991). Although it seems relatively simple, the task of selecting a good set of objects is not a straight-forward activity, the main reason being that there is no general consensus on the best way to identify the right objects. Ideally a requirements analysis will have been completed prior to object identification, during which information regarding functional and non-functional requirements has been gathered. This information is then used as a basis for deriving the initial set of objects to be included in the system. Various methods of analysis have been described in Section 3.2, all of which may form the starting point for this task. Identifying the right objects is highly domain-specific. Goldberg (1984) observes that the "appropriate choice of objects depends, of course, on the purposes to which the application will be put and the granularity of information to be manipulated". It is the policy of the proposed development method not to offer a single prescription to follow for selecting objects. It may be advisable and desirable to use a combination of the techniques already suggested for finding objects. Most would also agree that the task of object identification still requires a balance of intuition, experience and creativity.

Objects that are derived from analysis will generally be problem-domain or real-world entities. Thomas (1989) notes that while conducting the analysis, "if the domain expert talks about it, then the abstraction [or entity] is usually important" and should be included as an object in the system. Problem-domain objects form the framework of a model of real-world activities that is independent of the functionality the system has to provide. This model serves as a basis for communication with the user, defines the terms for system functionality and implicitly circumscribes a set of possible functions. Because the model is more stable than the functions, problem-domain objects tend to be stable and will normally survive any subsequent changes to the functionality of the system and remain in the architecture of the system. This principle of modelling reality is shared by the object-oriented paradigm and the operational approach. Additional objects may also be required which are not necessarily part of the problem domain but which are essential for the functions of the system; these extra objects belong to the solution domain. For example, in a bank, obvious objects in the domain that must be considered are customers, accounts, deposits and withdrawals. A system for the bank must also take into account other objects such as a database object to store information about customers and their accounts. Stroustrup (1986) suggests that after a set of objects has been selected, the decision to include them must be evaluated by asking questions such as: "How are objects of this class created ? Can objects of this class be copied and/or destroyed ? What operations can be done on such objects ? If there are no good answers to such questions, the concept probably wasn't 'clean' in the first place, and it might be a good idea to think a bit more about the problem and the proposed solution instead of immediately starting to 'code around' the problems".

When a set of objects has been obtained, it is necessary to decide on the level of abstraction of each; for example, some objects may become the child objects of others. As more details of the system are developed, examining the pattern of interactions amongst objects and implementing an object may suggest improvements or changes to the boundaries of objects, or expose other objects required in the system. New objects identified as a result of implementing higher abstraction objects are less likely to be problem domain entities and exist solely to support the functionalities of more abstract objects.

The product of the object identification step is a list of candidate objects that should be implemented in the system. At the highest level of abstraction, the objects identified are recorded in the notation using an *object interaction diagram*; the object interaction diagram is used to specify the system function aspect of an object-oriented system's dynamic behaviour in terms of the message interactions of the system's objects (at the highest level of abstraction). In the object identification step, the object interaction diagram contains only a group of objects that form the framework of the system; interactions amongst the objects are added in the diagram in the next step in the development process. Objects belonging to lower abstraction levels and which form the attributes of other objects are recorded in *object attribute diagrams* of their respective parent objects. Object attribute diagrams are used in the third step of the development process (described in Section 5.5) when the detail of each object is determined. Objects are represented in an object interaction diagram by the following symbols (Figure 5.1):

Figure 5.1 Object symbols used in an object interaction diagram.

(a) is used to represent a single object, and (b) represents two or more similar objects. For example, consider a library system consisting of books, members, and the library itself. The objects present would be a `library` object and multiple `member` and `book` objects, which will appear in an object interaction diagram as shown in Figure 5.2.

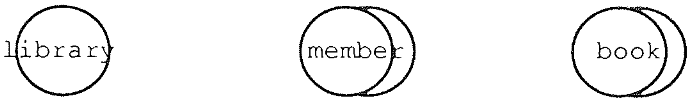


Figure 5.2 Objects in a library system's object interaction diagram.

5.4 Establishing Object Interactions

"An object by itself is intensely uninteresting. Objects contribute to the behavior of a system by collaborating with one another" (Booch, 1991). The second step in the development process is to determine the message interactions — using relationships — amongst the objects in the object interaction diagram created in the previous step, which involves deciding what objects an object may send messages to. The message interaction relationships define the *structure* within which objects work together to provide the system behaviour that would satisfy the requirements of the problem.

An example of such a structure housing and regulating object interactions is the graphical user interface mechanism used in the Smalltalk-80 programming system known as the Model-View-Controller (MVC) paradigm (Krasner & Pope, 1988). In this paradigm three main objects cooperate with each other to produce an image in a window: a controller, a view, and a model whose detail is to be displayed in the view.

The controller sends messages to its view and model objects when it receives instructions (e.g., through menu selections or mouse button input) to display contents of the model in the view or to initiate changes to the model. The view also sends messages to the model to obtain the relevant information for creating an image to be displayed. However, the model object is entirely decoupled from both the controller and view and cannot send any message to them directly.

Obviously, the structure of a system could have different configurations, where objects may be connected in different ways to produce the same functionality of the system; a design decision has to be made as to the most suitable form this structure should take. Once a particular structure is chosen, "the work is distributed among many objects" (Booch, 1991) by defining the protocol of these objects to include the appropriate messages required to achieve the functions of the system. Establishing the message architecture of the system first helps to facilitate the next step in the process of defining the semantics of objects by suggesting the messages that need to be included in the protocol of the appropriate objects.

The system's interactions with its environment also need to be considered at this point, in terms of which objects can receive messages directly from the system's environment. These messages must also be provided in the objects' protocols. Thus, the object interaction diagram serves as a model of the system operating within its environment. When an interface object receives a message from the system's environment, the object's processing may result in messages being propagated to other objects throughout the system's structure.

Results from requirements analysis could facilitate the process of determining visibility relationships amongst the objects in an object interaction diagram. How this proceeds depends on the particular analysis method used. Abbott (1983) suggests using verbs appearing in an English description of the problem to identify message interaction between objects; e.g., the statement "a member borrows a book from the library" would suggest an interaction between the objects `member` and `library`. In data flow diagrams produced from Structured Analysis, processes can also suggest message interactions. Using object-oriented analysis methods (some of which are described in Section 3.2) should simplify the task of identifying message interactions as they address the different relationships between objects in the problem domain.

The structure of message interactions amongst objects is expressed in an object interaction diagram as a network in which the objects communicate via *message channels*. Each message channel is unidirectional and carries messages from its sender object to its connected receiver object. A message channel is shown as a directed arc indicating the direction of flow of messages, with a small rectangle appended to it, denoting an unspecified message being sent along the channel. An object which interfaces with the problem environment is identified by attaching an *external entity object* to the object; the external entity provides environment input to the system by sending messages to its linked object. External entities are represented by shaded object symbols to distinguish them from system objects, and to signify that internal details of external objects are not required to be defined. Each external entity in an object interaction diagram has the same name as the system object to which it is linked. The MVC example described previously could be represented as shown in Figure 5.3. The figure shows an example in which the model receives messages from one controller and view[1] :

[1] In Smalltalk more than one controller and view pair can share the same model object, each concerned with a different aspect of the model.

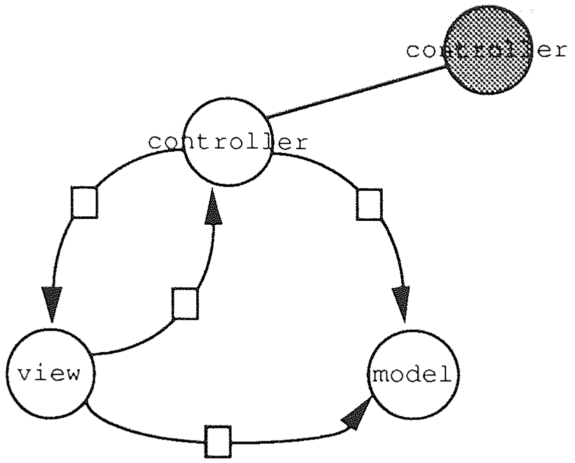


Figure 5.3 Object interaction diagram for the MVC paradigm.

5.5 Determining Object Semantics

Having established the topology of object message interactions in the system, the next step in the development process is to determine the detail of each object in the object interaction diagram. The activities required in this step include building the parent-child hierarchy of each object, and composing the protocol of the object. As described in Section 2.2, the parent-child hierarchy describes the containing relationships of an object with its attribute objects in its state. Attribute objects collectively describe the characteristics of the object and contribute to the functionality of their parent object; the parent object is, therefore, an abstraction of its attribute objects. For example, a car object would contain attribute objects such as wheels, steering, engine, etc. The use of an object-oriented analysis method such as Coad & Yourdon (1990, 1991) would undoubtedly facilitate this activity of determining object semantics as the characteristics of an object identified during the method would suggest appropriate object attributes.

A metric which could help in constructing the parent-child hierarchy is *cohesion*, which "measures the degree of connectivity among the elements of a single module" (Booch, 1991) — in the object-oriented paradigm, a module and its elements correspond to a parent object and its state objects respectively. Booch considers the least desirable form of cohesion to be *coincidental cohesion*, in which completely unrelated abstractions are contained in an object; as an example of a badly constructed parent-child hierarchy, consider an object containing attributes such as `numSunWorkstations` and `carRegistration` which do not seem to describe any coherent concept/abstraction. *Functional cohesion*, in Booch's opinion, is the most desirable, in which the attribute objects would all cooperate to provide the required behaviour of their parent object. Perhaps another relevant form of cohesion associated with an object would be *descriptive cohesion* whereby each attribute describes a characteristic of its parent object, e.g., attributes such as `name` and `address` describe a person object. The process of abstraction (see Section 2.2) must be applied here so that only the attributes relevant to the central purpose of an object in the system should be considered. For example consider a `libraryMember` object; an attribute such as `religion` is of no interest to a library system and should not be included in the `libraryMember` object's parent-child hierarchy.

All objects in an object interaction diagram are 'colleagues' at the same level of abstraction. When an object in the diagram needs to send messages to another object in the same diagram, the sender object needs to be aware of the identity of the receiver object. The receiver object in this case does not conceptually form a part of the state of the sender object, but is an object with which the sender object has a close association. An object remembers the identities of its colleagues as part of its state; hence, an object's state contains the identities of other objects it needs to communicate with, as well as the object's attributes. Thus, it is necessary to include in an object's parent-child hierarchy the identities of objects to which the object is connected by message channels.

In designing the representation of an object, a knowledge of existing classes is important as "they serve as more primitive classes and objects upon which we compose all higher level classes and objects" (Booch, 1991), and hence could influence how we choose to represent the state of an object. The object-oriented process is, therefore, not strictly top-down as explained in Section 3.1. Wirth (1986) also states that "the choice of representation is often a fairly difficult one, and it is not uniquely determined by the facilities [i.e., existing classes] available. It must always be taken in light of the operations that are to be performed". Keene (1989) mentions a common dilemma of deciding either to store an attribute value or to compute it using other values. He illustrates this problem using the example of a `cone` object with the attribute `volume`, and asks whether `volume` should be stored as an attribute value in `cone`'s state, or whether its `volume` should be calculated using the `height` and `radius` attributes of the cone. In such situations, the choice between time and space efficiency needs to be made. There are also trade-offs between containing relationships and using relationships. "Containing rather then using an object is sometimes better because containing reduces the number of objects that must be visible at the level of the enclosing object. On the other hand, using is sometimes better than containing because containing leads to undesirable tighter coupling among objects. Intelligent engineering decisions require careful weighing of these two factors" (Booch, 1991).

Object Definitions

An *object attribute diagram* is used to represent the parent-child hierarchy of an object, revealing the underlying structure in the object consisting of its state objects and its colleagues with whom it communicates via message channels. An object attribute diagram is useful in highlighting additional objects that are needed in the systems and which therefore need to be investigated further. An object's attribute diagram is also required in the next step in the process, when a class definition for the object must be determined and implemented if an existing class cannot be found which provides the required implementation. All child objects appearing in the object's state in its attribute diagram become instance variables in the class definition. A rectangle connected to the object is used to denote the encapsulated state of the enclosing object containing its attributes. In the MVC example described earlier, the object attribute diagrams of the `controller` and `view` objects may be represented as shown in Figure 5.4. The multiple object symbol is used in each case to indicate that the controller and view objects may contain numerous menus and subviews respectively. A multiple object symbol in an object diagram indicates that the class implementing the object will necessarily use a `Collection` object (e.g., `Array`, `Set`, etc) as an instance variable of the class.

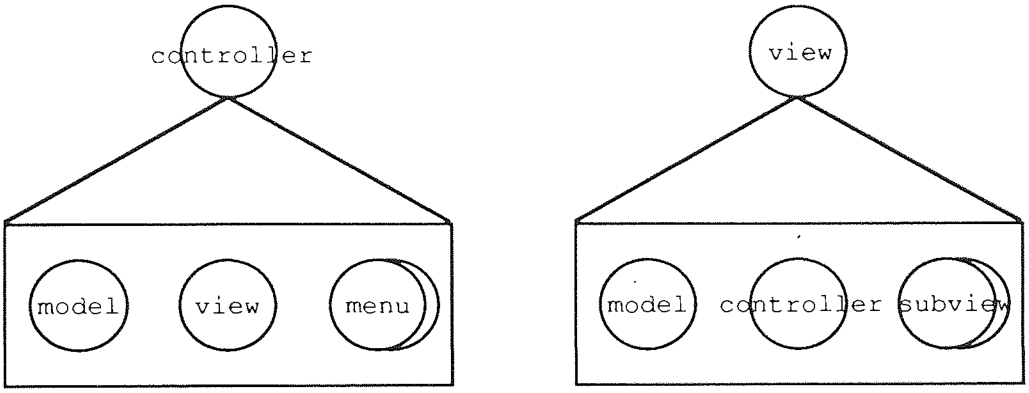


Figure 5.4 Object attribute diagrams of `controller` and `view`.

The second activity of composing the protocol of an object involves identifying the set of messages which may be sent to the object and the operations that may be performed on the object. The different analysis methods mentioned in Section 3.2 each suggests a different approach to message identification. Lippman (1989) and Booch (1991) suggest different ways of categorising the types of operation on an object (see Section 2.2); considering each of these categories would suggest some of the basic messages which should be included in the object's protocol. Pun & Winder (1989) observe that although functional decomposition from Structured Design is often dismissed in relation to object-oriented methods, functionally decomposing messages could often help to uncover further messages that are needed in an object's protocol; messages thus derived would probably be 'private' messages (Smalltalk-80 terminology) that are used only for the implementation of other messages. Also, studying an object interaction diagram and deciding what messages may be sent to an object on each in-coming message channel is a useful way of working out the object's protocol.

Object Life Histories

As the system receives input and executes its function, objects in the system are involved in receiving messages and then progressing through the phases in their own life history. In the notation, the life history of an object is represented using an *object life history diagram.* The life history aspect of dynamic behaviour in an object-oriented system is therefore represented by the object life history diagrams of the objects in the system. In their analysis method Coad & Yourdon use determination of entity life histories to help in identifying messages that apply to entities. Booch suggests writing a script for the life of each object starting from creation to destruction. Working out the life history of an object with respect to the order of messages that may be sent to it, therefore, is useful in identifying any missing messages that make up the life of the object. The object's life history could also help to clarify the contents of messages in the next development step when each object and its messages are implemented by a class and methods in the class.

Like entity life history diagrams and process structure diagrams in SSADM and JSD, an object life history diagram consists of three basic elements: sequence, iteration and selection. A sequence is composed of a list of messages which must be sent to the object in the strict order they appear in the sequence, without any backtracking or repetition. An iteration is a sequence which may be repeated an unspecified number of times, or not occur at all. A selection, on the other hand, is a group of messages which are mutually exclusive, i.e., only one message in the group may be sent to the object. An object life history diagram is a tree structure with messages as its leaves and the object it describes at the root. The main branch(es) of a life history tree form a sequence ordered from left to right; this sequence could of course, for example, contain a single message, iteration or selection. Messages are denoted by rectangles, each containing the *selector* (or name) of the message. Iterations and selections are represented as subtrees; the root of a subtree is a rectangle annotated by the symbol '*' or 'O' (an upper-case letter o) indicating an iteration or a selection respectively. A sequence may include iterations and selections if required conceptually to reflect the life of the object the sequence describes; similarly, an iteration could contain selections and other iterations. Figure 5.5 is an example of a valid object life history diagram.

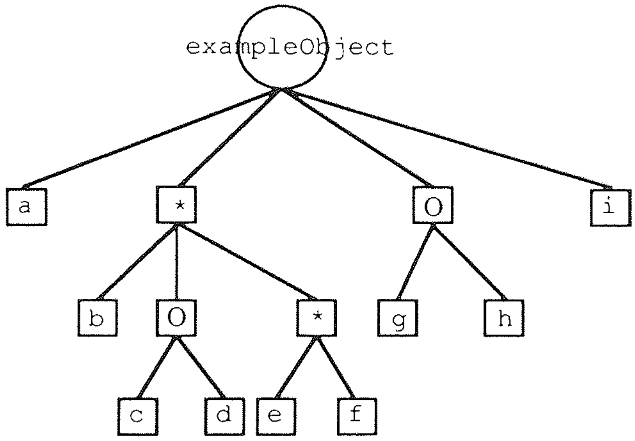


Figure 5.5 An example object life history diagram.

The sequence of the life history of object `exampleObject` consists of the message `a`, an iteration, a selection, and the message `i`. The iteration in the sequence contains the message `b`, a selection (of messages `c` or `d`) and an iteration (of messages `e` and `f`). The selection in the sequence comprises the messages `g` and `h`. Two possible sequences of messages received by the object `exampleObject` may be (`a`, `b`, `d`, `e`, `f`, `e`, `f`, `g`, `i`) and (`a`, `h`, `i`).

A selection can also contain iterations and other selections. The only restriction on a selection containing a selection subtree is that the subtree should not be connected directly to the higher-level selection (e.g., Figure 5.6 (a)); this is because a directly connected selection subtree is redundant as its alternatives could become the options in the higher-level selection as illustrated in Figure 5.6 (b). The selection subtree (a) may be replaced by the simpler description in subtree (b).

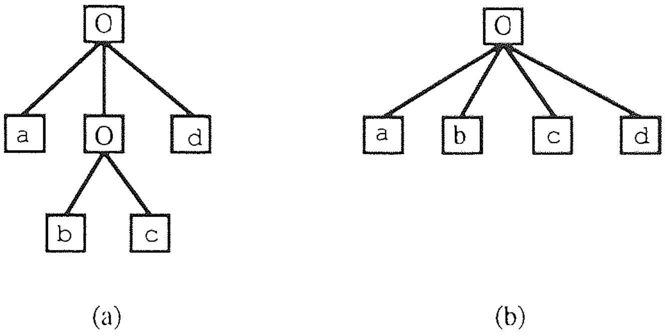


Figure 5.6 Simplifying selection in object life history diagram.

As an example, consider the life history of a `libraryBook` object in Figure 5.7 which may be compared with the JSD process diagram in Figure 4.3. Notice that the actions `Out Circulate` and `Deliver` in the JSD diagram have been omitted. This omission has happened because in JSD each action is regarded as atomic, but this restriction does not apply in this development method; the implementation of the message `swapScheme` is considered to involve sending the messages `outCirculate` and `deliver` to the `libraryBook` object.

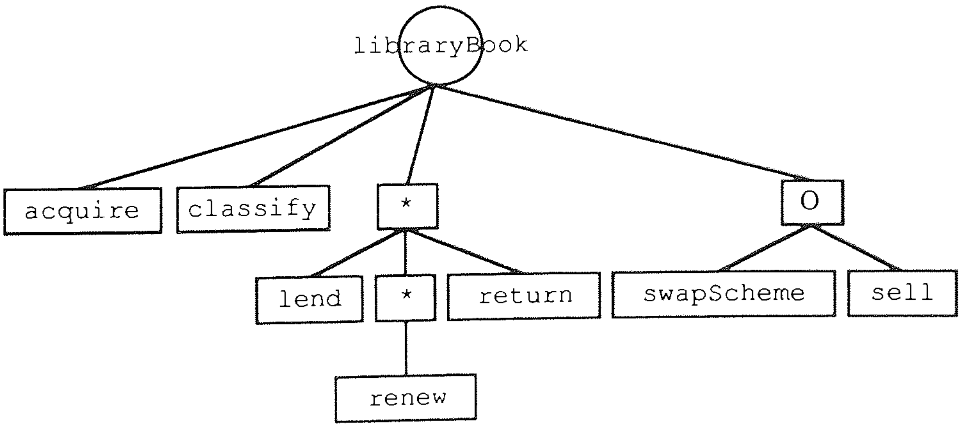


Figure 5.7 Object life history diagram of a `libraryBook`.

Certain messages are not constrained by the order in which they are sent to an object; such messages are usually inquiry messages used to obtain information about the object and do not contain the side effect of altering the state of the object. These unconstrained messages therefore are not included in the object's life history diagram.

Three metrics may be considered in relation with constructing the protocol of an object: *sufficiency*, *completeness*, and *primitiveness* (Booch, 1991). An object is sufficient if it possesses enough messages in its protocol to allow meaningful and efficient interaction, otherwise it is useless. Consider a `stack` object which can receive the `pop` message for removing the `stack` object's top item; the `stack` object's protocol would be adequate only if a `push` message is also included in the protocol so that items may also be added to the stack. To be complete, an object must have a protocol which provides all messages meaningful to the object. Sufficiency implies a minimal protocol, whereas a complete protocol must be general enough so that the object would be generally useful to any client. However, in the latter case, the object's protocol can become unnecessarily complicated if every conceivable message is included. Moreover, the operations of some complex messages may sometimes be achieved by using several more primitive messages, in which case these complex messages may be left out of an object's protocol. The third metric, therefore, suggests that primitive messages only should be included in an object's protocol. Primitive operations are those which can be implemented only by having access to the state of an object. For example, the `push` message of a `stack` object is primitive because it needs to manipulate the state of the `stack`. On the other hand, a message `push4Items` to add four items onto the `stack` is obviously not primitive since its purpose may be achieved by sending the `push` message four times.

5.6 Implementing Objects

When the semantics of objects have been established, these object details must be implemented as classes since each object is an instance of a class. An object may be instantiated from an existing class present in the target environment (either a pre-defined class provided in the environment, or a user-defined class implemented earlier) which provides the required definition for the object's state and protocol. If no suitable class can be found, a new class definition has to be created for the object. The issue of inheritance must be considered since each new class must be defined as a subclass of an existing one. Johnson (1988) observes that "software reuse does not happen by accident, even with object-oriented languages. System designers must plan to reuse old components and must look for new reusable components". When creating a class, existing classes must be examined to see if a suitable superclass can be reused; otherwise, it may be necessary for a new inheritance chain to be constructed to accommodate the new class. Effort must be made to design an inheritance structure that would be useful for future reuse; abstract classes could be used to abstract useful features that are likely to be inherited. Pun & Winder (1989b) describe a manipulation process called "inheritance factorisation", based on formal algebra, which could help in the design of a class hierarchy by suggesting inheritance relationships and abstract classes required by new classes.

An inheritance hierarchy may be wide and shallow, narrow and deep, or balanced (Lea, 1988). Lea comments that a wide and shallow structure contains a majority of free standing classes that can be mixed and matched, while a narrow and deep structure contains classes that are related by common ancestors. There are advantages and disadvantages to either approach. Free standing classes represent loose coupling, but they may not exploit all the commonality amongst the classes. On the other hand, classes that are closely related do exploit commonality, but in order to understand a particular class, it is usually necessary to understand the meaning of all the classes it inherits from. The appropriate shape of a class structure is highly problem-dependent. Consideration must also be given to the use of inheritance versus the containing relationship. For example, should the class `Car` inherit or use the classes `Engine` and `Wheel` ? In this particular instance, it may be obvious that the containing relationship would be more appropriate where instances of `Engine` and

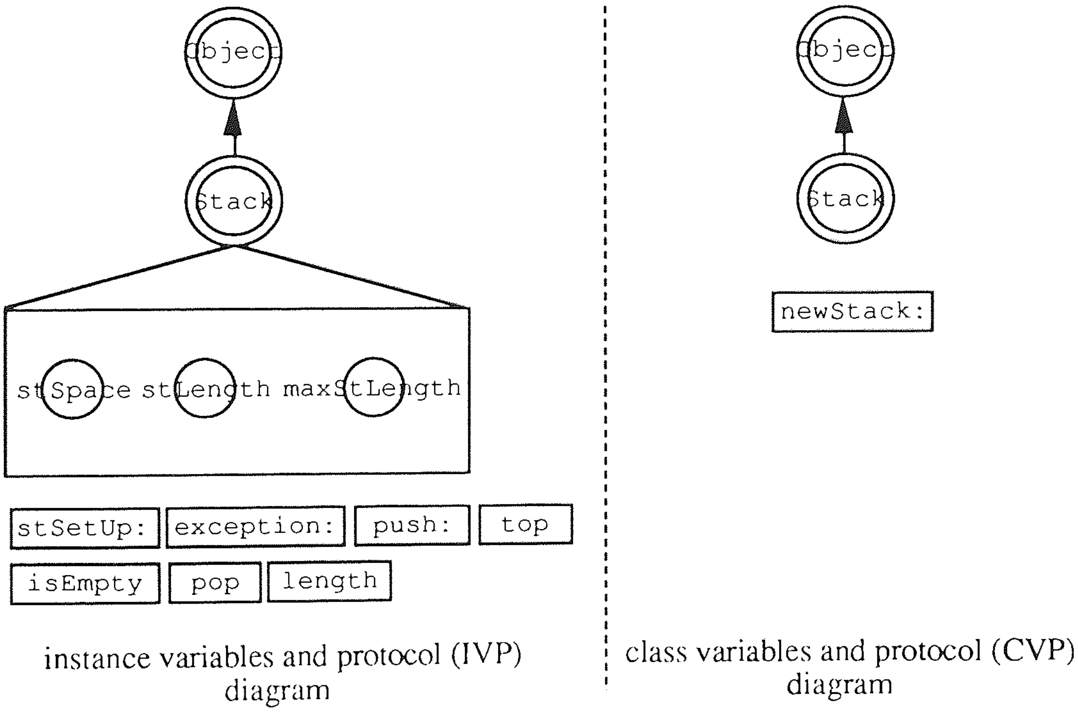
Wheel form part of the abstraction of a car. Meyer (1988) suggests that between two classes A and B, "inheritance is appropriate if each instance of B may also be viewed as an instance of A. The client [containing] relationship is appropriate when every instance of B simply possesses one of more attributes of A".

Class Definitions

A class is defined using a *class definition diagram*. As described in Section 2.2, a class's definition comprises both its instance variables and protocol, and its class variables and protocol; these two sets of information are contained in two separate diagrams — an *instance variables and protocol* (IVP) *diagram* and a *class variables and protocol* (CVP) *diagram* — that together compose a complete class definition diagram. An IVP diagram shows a class's instance variables and messages, while a CVP diagram displays its class variables and messages. The IVP and CVP diagrams of an example Stack class is shown in Figure 5.8. In the IVP diagram, the Stack class is denoted by a double circle which distinguishes it from an instance object, and represents a template from which its instances are instantiated. The instance variables stSpace, stLength and maxStLength are represented in the same way as state objects in an object attribute diagram; a rectangle connected to the Stack class symbol encapsulates the instance variables denoted by single object symbols. Each instance message is shown as a small rectangle containing the selector of the message, e.g., push:, pop and top, and these are located outside the encapsulation enclosure of the instance variables. The clear segregation of the messages from the instance variables shows that the messages form the shared visible interface of the instances, while the instance variables are the protected details of the instances.

Inherited variables and protocol form an integral part of a class's definition. An IVP diagram shows this ancestral detail by including the class's immediate superclass in the diagram. The superclass is represented by a class symbol above the class, and the inheritance relationship between them is denoted by an arrow from the class to its superclass. Wilson (1990) suggests it is important that the direction of an arrow representing an inheritance relationship should signify that the subclass is referring to the definition of its superclass, not vice versa, and the existence of the subclass is anonymous to its superclass. Figure 5.8 shows that `Stack` is a subclass of `Object`.

A CVP diagram is represented in the same format as its IVP counterpart. The CVP diagram of `Stack` in Figure 5.8 shows that it has one class message (`newStack:`) but contains no class variable.



instance variables and protocol (IVP) diagram

class variables and protocol (CVP) diagram

Figure 5.8 Class definition of class `Stack`.

For each class, only the immediate superclass is shown in its class definition diagram to avoid complexity. A separate *class inheritance diagram* is required to show the total inheritance chain of a class, leading from the class itself through all its superclasses to the root of its inheritance chain — the class `Object`. `Object` is the superclass of all other classes, and provides the basic default behaviour common to all objects, such as instantiation, copying and printing. The class symbol is used to denote each class in the inheritance diagram, and an inheritance arrow links each subclass to its superclass in the chain. Because inherited characteristics are not enumerated in class definition diagrams, the class inheritance diagram serves as an important guide to accessing the full definition of a given class.

Method Definitions

The message in each class definition must be implemented by a method. A useful guideline for the definition of a method is provided by the Law of Demeter, which states that "the methods of a class should not depend in any way on the structure of any class, except the immediate (top-level) structure of their own class. Further, each method should send messages to objects belonging to a very limited set of classes only" (Sakkinen, 1988). The aim of this rule is to encourage creating classes that are loosely coupled so that understanding the characteristics of a class is not dependent on knowing the details of other classes. Classes are also more robust and easier to modify as their implementation details are encapsulated.

It is sometimes necessary to decompose a complex method and contract out behaviour to one or more other methods. Spreading behaviour across several methods leads to a more complicated class protocol, but the methods are simpler. On the other hand, if a single monolithic method is used, a more complicated method may result while keeping the protocol of the class simple. Meyer (1987) observes that "a good designer knows how to find the appropriate balance between too much contracting, which produces fragmentation, and too little, which yields unmanageably large modules".

Working out the meaning of a method may result in a better understanding about the life history of objects of a class, so modification to the life history of objects may be necessary. This may in turn uncover more messages required in the object's protocol which will need to be implemented in the object's class. During the implementation of a class, it is necessary to determine the set of classes the class uses. The instances of this set of classes make up the state of each instance of the client class. Unless the state objects are trivial and may be instantiated from available classes, the steps of the development process must be repeated at this point and applied to designing the details of these state objects. The process is therefore highly iterative and stops only when all the objects and classes of the system have been discovered and implemented by existing classes.

A method is defined in the notation using a *method definition diagram*, which contains the executable semantics in a specification. The structures and semantics of constructs used in a method definition diagram are based on the Smalltalk-80 language. A method definition diagram consists of a declaration section in which formal parameters and local variables required in the method's definition are declared. There are three types of message: *unary*, *binary* and *keyword*. Formal parameters in binary and keyword messages are variables which contain references to actual parameter objects; each formal parameter is denoted by an object symbol. A unary message does not need any parameter, an example being the `pop` message of a `stack` object. A binary message takes a single parameter, and is composed of one or two non alphanumeric characters; examples for `number` objects are +, -, <=, etc. A keyword message requires one or more parameters. The selector (name) of a keyword message consists of one or more keywords, each preceding a parameter; each keyword is an identifier with a trailing colon. An example keyword message for the `stack` object is `push: 30`, which adds the number `30` to the `stack`. The three types of message are represented in the following format in their method definition diagram (Figure 5.9):

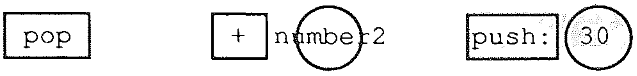


Figure 5.9 Examples of unary, binary and keyword messages.

Local (or temporary) variables required in a method definition are declared in a similar manner to instance variables in a class definition diagram. Local variables are represented by object symbols enclosed in a rectangle connected to the message representation in the declaration section of the method definition; the rectangle shows that the variables it contains are private to the method. Not all local variables that may be required in a method could be known when the method is first declared. When the details of the method are developed, the local variables needed will emerge and can be added to the declaration as they appear. As an example, consider an initial declaration of the method for the instance message `push:` belonging to class `Stack` (Figure 5.10):

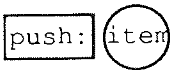


Figure 5.10 Declaration of the `push:` message.

A formal parameter `item` is required to refer to the actual parameter object to be added to a `stack` object; at this stage, no local variable is known yet.

In each method, messages are sent to other objects and/or other classes to carry out its task. Objects and classes are represented by their usual symbols in a method definition diagram. Other objects that may appear in a method definition include literal objects and objects referred to by variables. Literal objects are constant values and are represented by literal expressions. The set of literal objects supported are character, string, symbol, number, and array. Explanations and examples of each class of literal object are given below (Figure 5.11).

Class	Explanation	Literal expression
Character	- a symbol of an alphabet	- denoted by a dollar sign e.g. `$a  $1  $A  $3`
String	- a sequence of characters	- delimited by single quotes e.g. `'hello world'`
Symbol	- a string used as a unique name	- preceded by a hash sign e.g. `#statusOK  #M6`
Number	- a numeric value	e.g. `-3  54`
Array	- a data structure containing literal values indexed by integers from 1 to the size of the array	- delimited by parentheses and preceded by a hash sign - each pair of literal values are separated by a space e.g. `#(1  'Me'  $A)` - an array containing a number, a string, and a character `#('a'  (-123)  #open)` - an array containing another array at index 2

Figure 5.11 Different classes of literal object provided.

Three basic types of statement make up the definition of a method: ***message expressions***, *assignment expressions*, and *return expressions*. A message expression refers to a message send and its receiver. A message expression is represented by connecting the message (and any parameters) to the object with a vertical line (see Figure 5.12). A variable is made to refer to an object via an assignment expression, which is denoted by an arrow linking the assigned value (an object) to the variable (see Figure 5.12). The object being assigned may be derived by one of three ways: it may be a literal object or another variable, or a message expression. The result of a message send is always an object; the receipt of a return value from a message expression indicates to the message sender that the response of the receiver is complete.

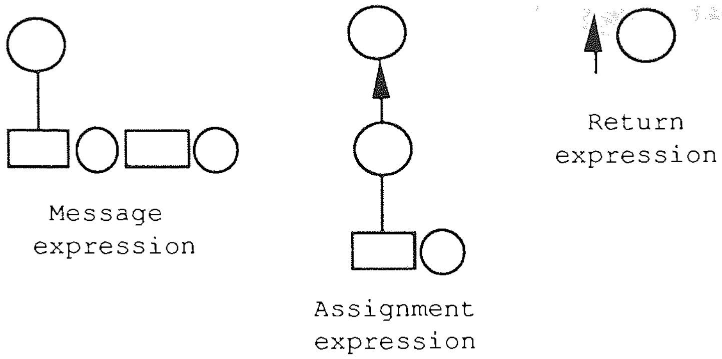


Figure 5.12 Examples of a message expression, an assignment expression, and a return expression.

Some messages are meant only to inform the receiver about something, e.g., the `stack` object message `push: 30` to add the number `30` to the `stack` object; in such cases the values of message expressions are not significant. On the other hand, if the sender is expecting a specific response from sending a message (e.g., the message `top` to return the top object in a stack), the method of the receiver needs to include an explicit return expression to return the correct object. If a return expression is missing in the method of a message, some default value will be returned automatically. A return expression is denoted by preceding the return value with a vertical arrow (see Figure 5.12). The return object may be derived in three ways:

— a literal object or the object referred to by a variable

— the value of a message expression

— the assigned object of an assignment expression.

Another type of variable exists in Smalltalk-80, known as a *pseudo-variable*. A pseudo-variable differs from a normal variable in that its value cannot be altered by an assignment. Some of these pseudo-variables are constants, i.e., they always refer to the same objects; important examples are `nil`, `true` and `false`, which refer to the default `null` object and the `Boolean` objects. The other pseudo-variables have values that change depending on their contexts; examples are `self` and `super`. The variable `self` always refers to the message receiver object executing the message's method in which `self` appears; therefore, in different invocations of the same method by different receivers, `self` refers to a different object in each instance. `super` also refers to the message receiver in a message's method, except that a message sent to `super` causes the search for the message's method to start in the immediate superclass of `super`'s class. Both kinds of pseudo-variables are supported, and are represented using the object symbol as shown in Figure 5.13:

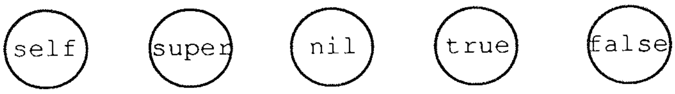


Figure 5.13 Pseudo-variables.

Control Structures and Blocks

A control structure determines the order of execution of statements in a method. The basic control structure is sequential in which statements are executed in sequence from left to right in the method definition diagram. Non-sequential forms of control are provided using *blocks*.

A block is an object which represents a sequence of statements. When execution reaches a block object, the statements contained in the block are not executed immediately (i.e., the execution is deferred); the statements are evaluated only when the block object is sent the unary message `value`. The result returned from sending the `value` message to a block is the value of the last statement in the block's sequence; an empty block will return the `nil` value. Like other objects, blocks may be assigned to variables; the only restriction is that a block must be assigned to a variable before a message can be sent to it. A block is represented in a method definition diagram by the symbol ┌──────┐ delimiting any statements the block contains, instead of the usual object symbol. Figure 5.14 shows an example block object containing a message expression, an assignment expression, and a return expression.

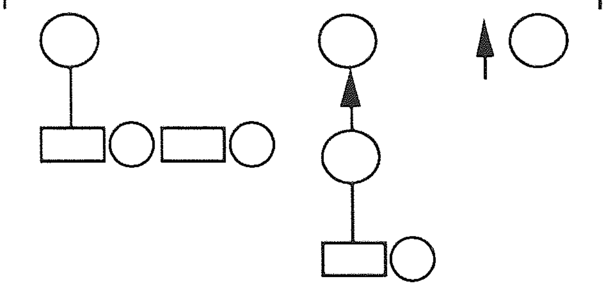


Figure 5.14 Example of a block object.

Two forms of non-sequential control structure are provided: *conditional selection* and *conditional repetition*. Conditional selection is similar to the `if...then...else` statements of languages such as C and Ada, and is provided by the messages `ifTrue:ifFalse:`, `ifFalse:ifTrue:`, `ifTrue:` and `ifFalse:` sent to the `Boolean` objects `true` and `false`; all these messages use block objects as arguments. Figure 5.15 shows the message `ifTrue:ifFalse:` being sent to a `Boolean` object referred to by the variable `booleanObject`. Conditional repetition is similar to the `while` and `until` statements in those procedural languages. Conditional repetition is achieved by sending the messages `whileTrue:`, `whileFalse:`, `whileTrue`, `whileFalse` and `repeat` to a block object, with a second block as argument if required. A block receiver of a conditional repetition message must contain statement(s) which evaluate to a `Boolean` object.

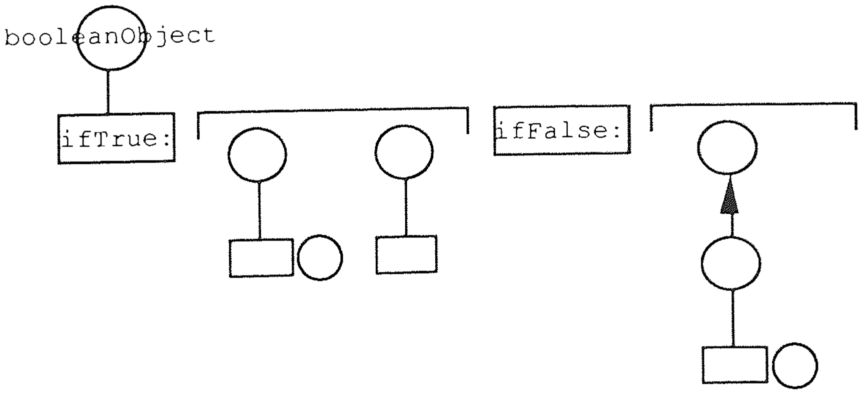


Figure 5.15 Example of sending the `ifTrue:ifFalse:` message to a `Boolean` object.

As stated earlier, a restriction on a block object is that it has to be assigned to a variable before a message may be sent to it. An example of using the whileTrue: message in a method definition diagram is shown in Figure 5.16, in which a block object is first assigned to the variable variableBlock before the message whileTrue: is sent to the block.

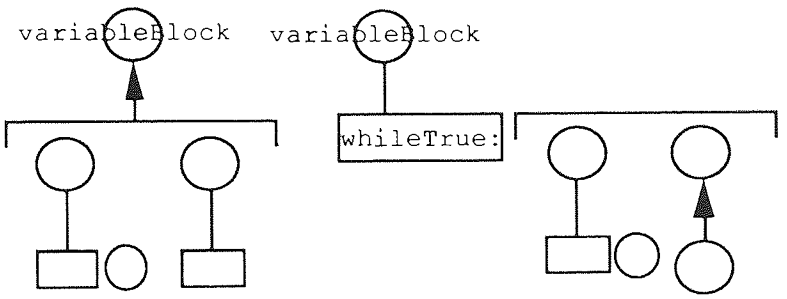


Figure 5.16 A block assignment and sending the whileTrue: message to the block.

To illustrate the definition of a method, the instance message push: of the class Stack is declared and defined using the method definition notation introduced, based on the Stack class definition shown in Figure 5.8. The instance variable stSpace uses an Array object to store the contents of a Stack object. maxStLength and stLength are Integer objects indicating the size of the stSpace array and the current number of items on the stack respectively. In the push: method (Figure 5.17), the conditional selection message ifFalse: is used to add the item object onto a Stack object only if the Stack object is not already full.

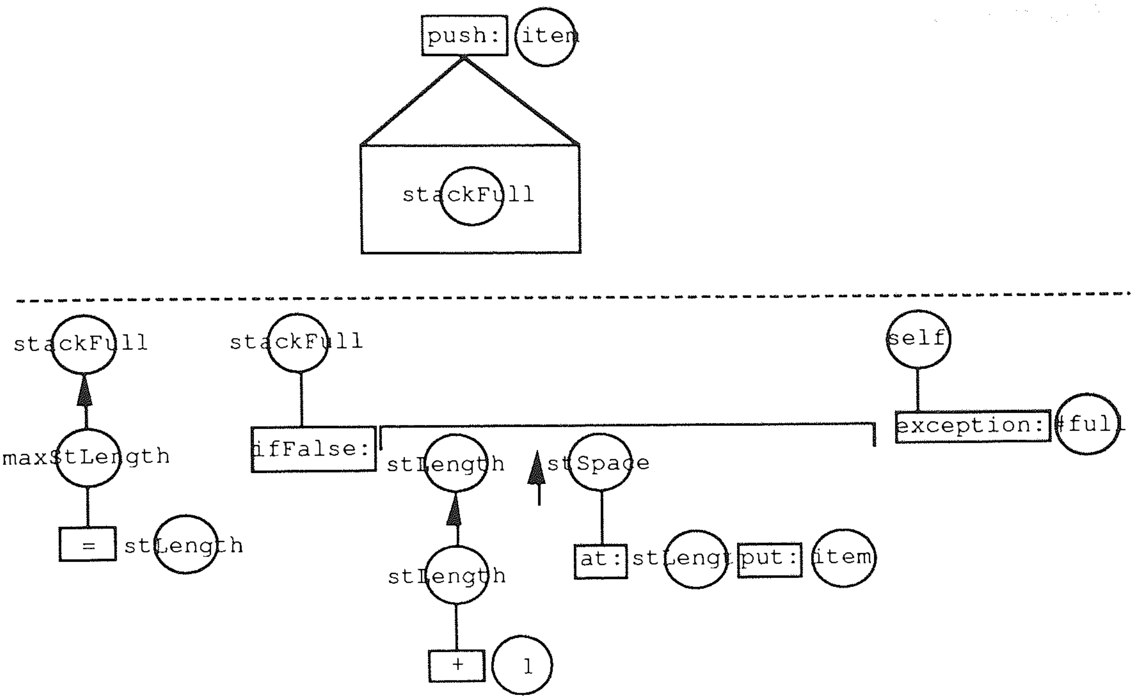


Figure 5.17 Method definition for `push:`.

5.7 Example

To illustrate the use of the notation introduced in the preceding sections, an example specification of an automobile cruise control system is described below.

Brief Problem Definition

The driver of a vehicle normally monitors the speed of the vehicle by checking the reading on the speedometer and moving the accelerator to keep the vehicle's actual speed close to a desired speed. The cruise control system performs this task for the driver by automatically maintaining the vehicle at the requested speed.

The cruise control system can operate when the vehicle's engine is switched on. When activated, the system stores the current speed as the desired cruising speed and maintains this speed by monitoring the speed shown on the speedometer and then setting the throttle to the required position. While cruising, the driver can instruct the system to accelerate, then stop accelerating and use the current speed as the new cruising speed. The driver can deactivate the system directly to stop cruising; switching off the vehicle's engine also has the effect of deactivating the system. The driver can also use the brake to temporarily disable cruising and slow down, and then request the system to resume cruising once the target speed is reached.

Specification

The first task is to identify the main objects and their interactions. The obvious objects that can be pinpointed so far from the problem description are the cruise control system, the engine, the speedometer, the brake and the throttle. The cruise control system needs to send messages to the throttle and speedometer to set the throttle position and to enquire the current speed respectively. The cruise control system also needs to send messages to the engine to check if it is switched on before cruising can operate. The engine and brake need to send messages to the cruise control system to inform the latter when the engine is switched off (and so cruising, if in operation, needs to be disabled), and when the driver is braking. These objects and their interactions are recorded in an object interaction diagram (Figure 5.18).

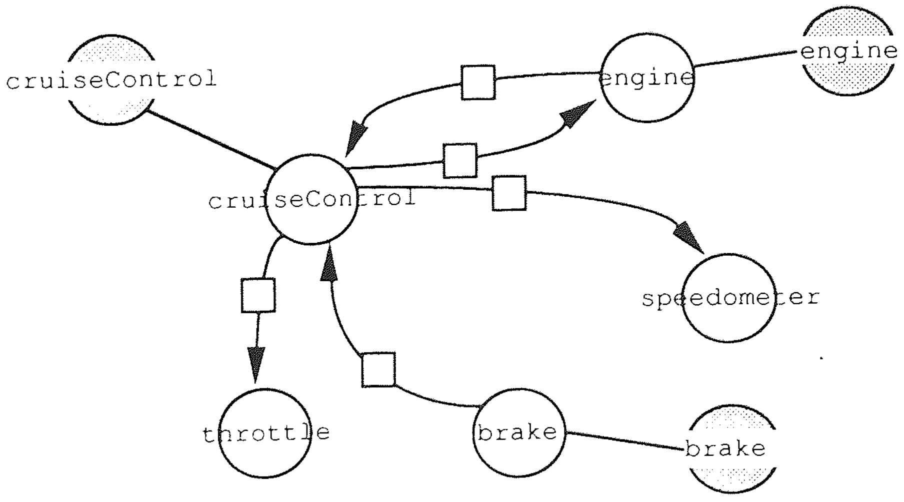


Figure 5.18 Object interaction diagram of the cruise control system.

Next, the driver's interactions with the system is determined to identify the interface objects. The driver can activate and deactivate cruising, and can instruct the system to accelerate or stop acceleration; the engine can be turned on and off, and the user may use the brake to slow down. Therefore, the interface objects are `cruiseControl`, `brake` and `engine`, each of which has its own external entity in the object interaction diagram in Figure 5.18.

Each object in the object interaction diagram has to be defined using an object attribute diagram and an object life history diagram. In examining the detail of the `cruiseControl` object, it is obvious that it's colleague objects are `speedometer`, `engine` and `throttle`. Other objects required in the state of `cruiseControl` would include `status` and `desiredSpeed`. These objects are shown in the object attribute diagram of `cruiseControl` in Figure 5.19.

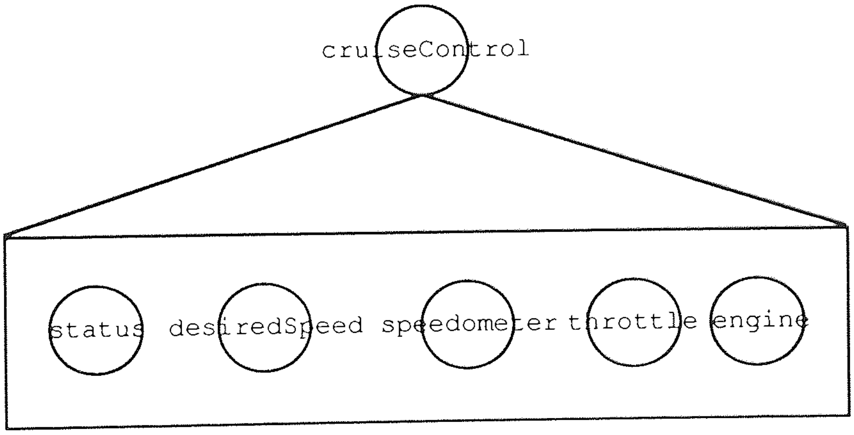


Figure 5.19 Object attribute diagram of the `cruiseControl` object.

The cruising function of the `cruiseControl` object may be activated and deactivated repeatedly; the life history diagram of `cruiseControl` shown in Figure 5.20 describes this with an iteration of the messages `cruise` and `stopCruise`. While cruising, `cruiseControl` may be instructed to accelerate and then stop acceleration an infinite number of times; this is represented by an iteration of the messages `accelerate` and `stopAccelerate` within the first iteration (of `cruise` and `stopCruise`). On receiving the `accelerate` message, `cruiseControl` must first disable cruising, hence the stopCruise message following the accelerate message. When instructed to stop accelerate, cruising must again be enabled so that the current speed is used as the desired speed, hence the cruise message following stopAccelerate. The behaviour of the brake object may cause cruiseControl to stop acceleration or disable cruising depending on its status. When the driver wishes to resume cruising, cruiseControl would start its operation after stopCruise or stopAccelerate.

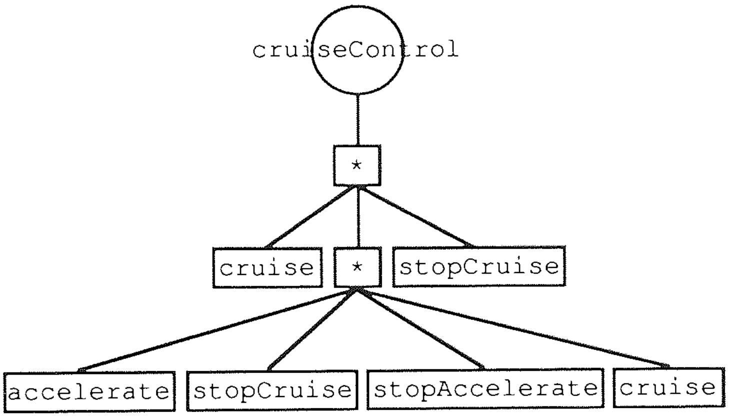


Figure 5.20 Life history diagram of the cruiseControl object.

Both the child objects status and desiredSpeed can be implemented by the pre-defined Smalltalk-80 classes Symbol and Integer and so do not need to be defined further. Colleague objects throttle, engine and speedometer would require definition with respect to their states and life histories. When all the system objects in the object interaction diagram have been defined, the next task is to determine the class implementation of the objects. The cruiseControl object cannot be implemented by an existing class in the Smalltalk-80 library; a new class CruiseControl needs to be defined for its implementation. Figure 5.21 shows the IVP diagram of the class definition of CruiseControl.

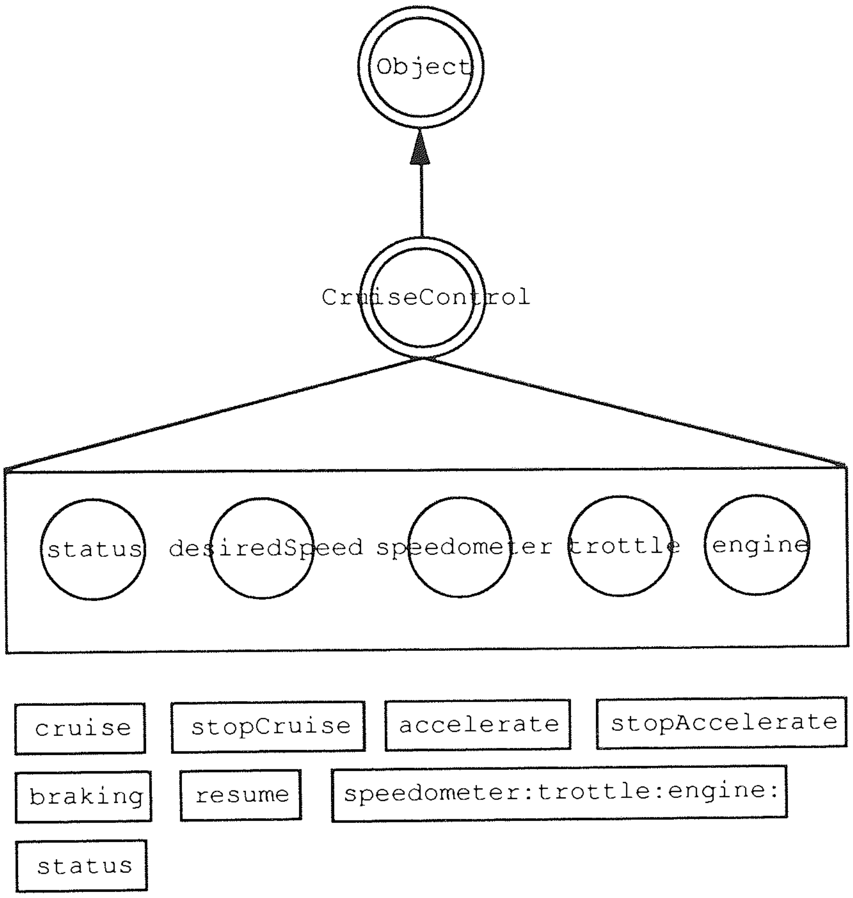


Figure 5.21 Class definition diagram of the class `CruiseControl`.

The instance protocol of the class includes a message to initialise the identities of the colleague objects (`speedometer:throttle:engine:`), and the messages `cruise`, `stopCruise`, `accelerate`, `stopAccelerate` and `braking`, etc. Each message is implemented by a method. Figure 5.22 shows the method definition diagram for the method `cruise`.

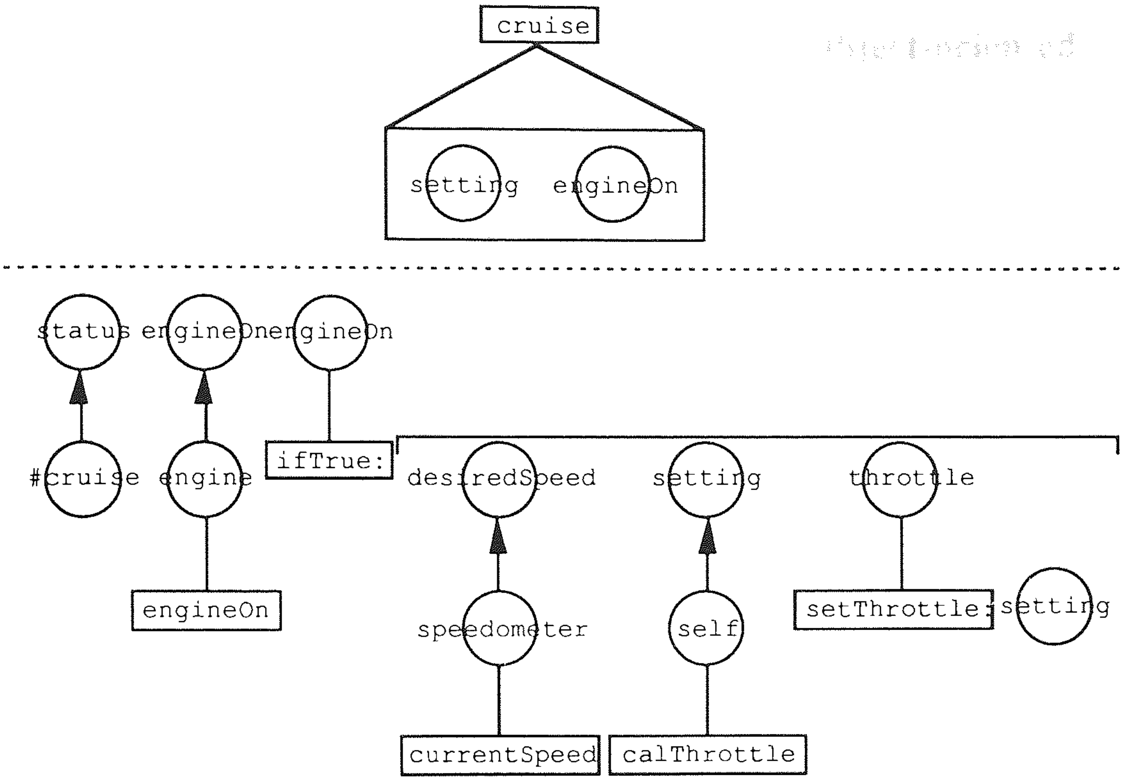


Figure 5.22 Method definition diagram of the method `cruise`.

The method first checks that the engine is switched on. The current speed is then used as the desired speed to calculate the required throttle setting. The throttle is then set to the required value to maintain the desired speed.

5.8 Summary

An operational specification notation for object-oriented systems has been introduced and the development process for using the notation has been described. A tool is required which can not only facilitate the graphical input of specification diagrams, but more importantly, must be able to execute a specification and also animate this execution. Chapter 6 describes the implementation of this tool to support the development process.

Chapter 6 Tool Support for Operational Object-oriented Development

6.1 Introduction

A tool has been implemented to support the operational approach to object-oriented development discussed in the previous chapter. The implementation has been carried out using the Smalltalk-80 programming language (version 2.5). The two main objectives of the tool are:

— to support the building of operational specifications of object-oriented systems, using the graphical notation introduced in the last chapter

— to execute a system described by an operational specification generated with the tool, and to animate the system's behaviour.

Two main components make up the tool and perform its two functions: the *editor* and the *interpreter*. The editor facilitates the input of the various diagrams that comprise a specification. As the elements of the specification's diagrams are added, internal data structures representing the details of the specification are gradually constructed. When a specification is complete (or partially complete), the interpreter may be invoked which can access the data structure representation of the specification and then animate the behaviour of the system graphically using the diagrams of the specification, while the system is executed. Execution of a system and the animation of its behaviour is described in Section 6.5.

6.2 The Editor

The editor provides a WIMP interface for the graphical input of specification diagrams based on the MVC paradigm of the Smalltalk-80 system. Each type of diagram is created using its own view (window). The controller for each view provides context-sensitive pop-up menus for selecting editing functions to be performed on a diagram. Depending on the position of the cursor in a view, different pop-up menus are opened to show the relevant operations that may be performed with respect to the element in the view's diagram selected by the cursor. The controller classes used by the editor are implemented as subclasses of the class `DoubleScrollController`[1] which enhances the scrolling facility provided by the Smalltalk-80 class `ScrollController` by allowing scrolling to be performed on a view in the horizontal as well as the vertical direction for large diagrams. The editor is implemented as a group of controller and view class pairs used for creating the windows for specification diagrams; these classes include:

- `SpecificationController` and `SpecificationView`
- `ObjectLifeCycleController` and `ObjectLifeCycleView`
- `SystemObjectController` and `SystemObjectView`
- `ClassDefinitionsController` and `ClassDefinitionsView`
- `MessageExpressionsController` and `MessageExpressionsView`.

The following sections give a brief overview of how each type of specification diagram is created in its view.

[1] This class has been implemented by Colin Lewis.

A *specification view* is used for creating an object interaction diagram of a specification. In this view a system object or an external entity is added to the diagram by selecting the option `add single system object` (or `add multiple system object`), or the option `add external entity` respectively in the *specification menu* in an empty area in the diagram. The user is prompted for the name of the object or entity with a `FillInTheBlank` dialog box. The cursor then adopts the form of an object symbol or an external entity, with the shape of a hand over the symbol — a *mobile symbol cursor*. The mobile symbol cursor is used by the user to move the symbol around in the view. The symbol is placed in the diagram when the user clicks the mouse button at the desired location. Figure 6.1 shows examples of the mobile symbol cursor for the object `cruiseControl` and its external entity, and Figure 6.2 shows a specification view and its pop-up menus.

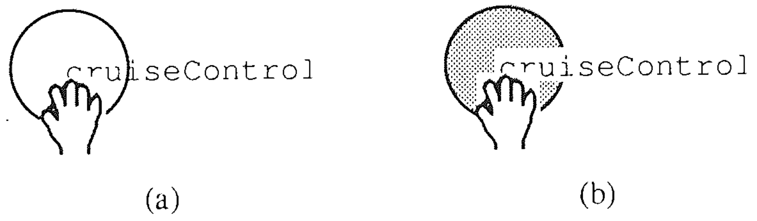
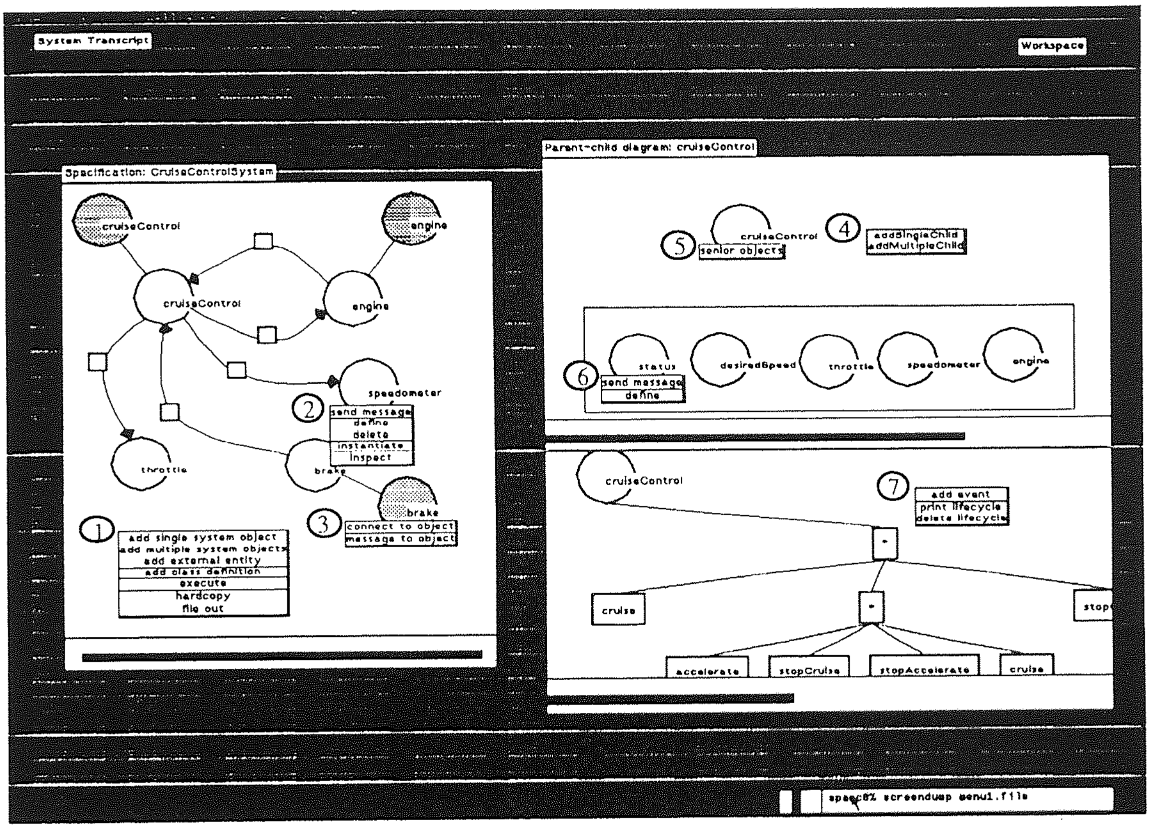


Figure 6.1 Mobile symbol cursor for (a) an object and (b) an external entity.

An external entity is connected to its system object by selecting the external entity and then the option `connect object` in the *external entity menu*. The connecting line leading from the external entity follows the movement of the cursor until the user clicks the mouse button to indicate the system object; if a system object is selected by the cursor the link between the external entity and the object is added in the diagram. A message channel between a sender object and a receiver object can be added by selecting the sender object and then the `send message` option in the *system object menu*. A message channel leading from the sender object tracks the movement of the cursor until the user clicks the mouse button twice, first to designate the location of the channel's message symbol, and then to indicate the receiver object; if a system object is selected as the receiver, the message channel is added in the diagram. By checking the validity of an external entity link or a message channel the editor helps the user to avoid creating a meaningless diagram.



```
spasc8% screendump menu1.file
```

Pop-up menus

1 specification menu
2 system object menu
3 external entity menu
4 object definition menu
5 parent object menu
6 child object menu
7 life history menu

Figure 6.2 A specification view (left) and a system object view (right).

When the `define` option is selected in a system object menu, a *system object view* (Figure 6.2) is opened to allow the user to create an object attribute (parent-child hierarchy) diagram and life history diagram for the selected system object.

Object Definition Diagrams and Life History Diagrams

A system object view consists of two subviews allowing the object attribute diagram and the life history diagram of an object to be viewed at the same time (Figure 6.2).

Pop-up menus are used in both subviews to help the user create the diagrams. In the object definition diagram, when the parent object symbol (placed in the diagram by default) is selected followed by the `senior object` option in the *parent object menu*, a list of names of all colleague objects which the parent interacts with is displayed. Selecting a colleague from this list causes the cursor to become a mobile symbol cursor which the user then uses to position the symbol for the colleague in the diagram. The encapsulation rectangle enclosing all child and colleague objects automatically adjusts itself to include the new symbol. A child object is added by selecting the `add child object` option in the *object definition menu* outside the parent object symbol. The user is prompted for the object's name and then its symbol is placed in the diagram in the same way as for a colleague object. Details of a child object (i.e., its object definition diagram and life history diagram) can be defined by selecting the child object and then the `define` option in the *child object menu*; a new system object view is then opened for the child object and its diagrams can be created in the same manner.

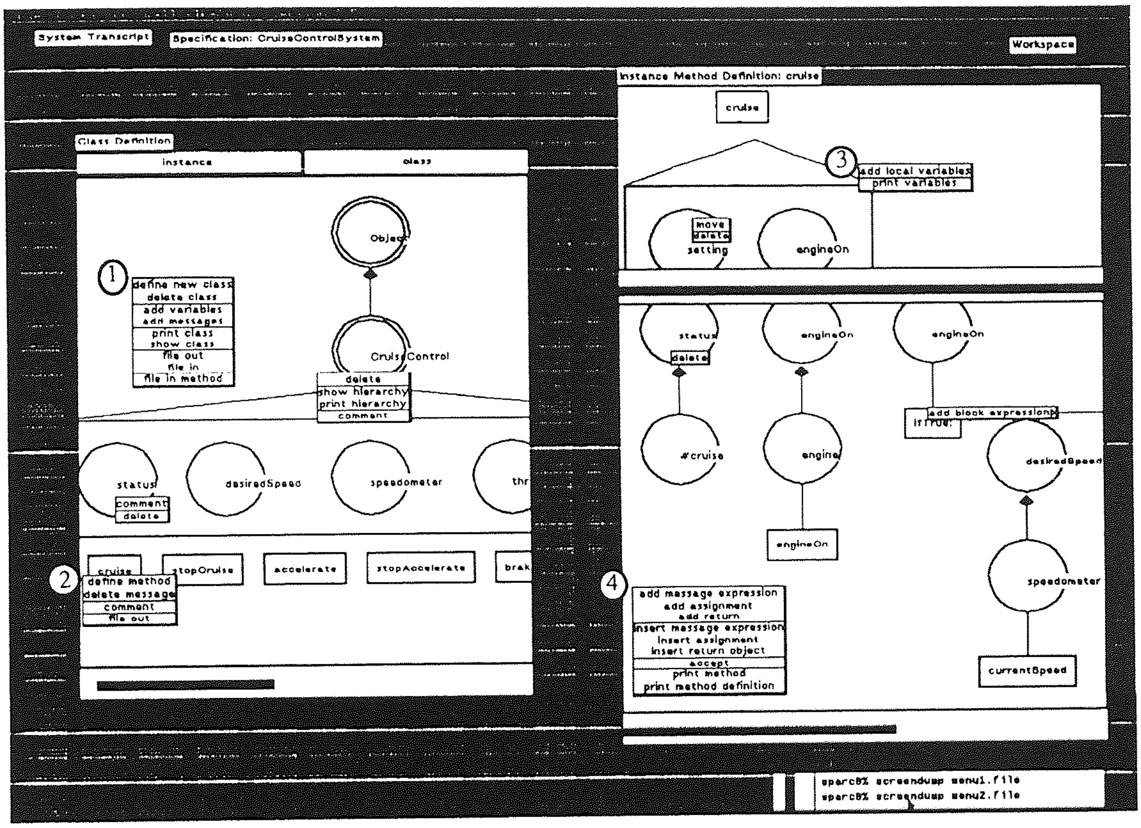
In the life history diagram, initially only the symbol for the owner of the life history is present. If the object does not need any constraint on its messages, its life history diagram will not need any modification. To add messages to the diagram, the user selects the `add event` option in the *life history menu* and is prompted for the event, which may be a message, an iteration or a selection. The symbol for the event is automatically placed in the diagram in the correct position ensuring that the diagram remains tidy. The user is then prompted for the next event to be added. This process continues until the user presses the return key when prompted, signalling that no more new event is required. The same process is used for building an iteration or a selection subtree; new events entered by the user are added as components of the subtree if the previous event was an iteration or a selection symbol, until the user presses return to complete the subtree. On completing a subtree, the process continues and the user is prompted for the next event to be added in the life history at the same level as the root of the iteration or selection subtree.

Class Definition Diagrams

A *class definition view* (Figure 6.3) is opened for the input of class definitions when the user selects the `add class definition` option in the specification menu. A class definition view consists of two boolean buttons and a subview. The class and instance buttons are used by the user to switch between looking at the IVP diagram and the CVP diagram of a class — they are similar to the class/instance buttons in the browser of the Smalltalk-80 environment. The subview displays either the IVP or the CVP diagram depending on the user's button selection — by default the instance button is selected when the class definition view is first opened.

To define a new class the user selects the `define new class` option in the *class definition menu* in an empty area in the subview. Assuming the instance button is selected, a new IVP diagram for the new class is displayed in the subview. The user is first prompted for the name of the class, followed by its superclass, and class symbols for both are placed in the diagram automatically in default positions connected by an inheritance arrow. The `add variables` and `add messages` options in the class definition menu can now be used to add instance variables and instance messages to the class respectively. In each case, a dialog box is provided for the user to enter a list of variables or messages. To complete a list the user selects the `accept` menu option in the dialog box, and symbols for the variables or messages are added in the diagram automatically. Default positioning is used to help the user generate a diagram quickly and to ensure its neatness. When the class button is selected, the same process is used for creating the CVP diagram of the class. There is no restriction on the order in which the two diagrams of a class definition are built, and the user can return to any diagram using the instance and class buttons to make alterations any time. The `show class` option in the class definition menu can be used to display any class defined previously.

To create a method for a message, the user selects the message and then the `define method` option in the *message menu.* A method definition view is then opened for the user to create the method's definition diagram.



```
sparc8% screendump menu1.file
sparc8% screendump menu2.file
```

Pop-up menus

1 class definition menu	3 method declaration menu
2 message menu	4 method definition menu

Figure 6.3 A class definition view (left) and a method definition view (right).

Method Definition Diagrams

A *method definition view* (Figure 6.3) consists of two subviews. The top subview displays the declaration section of a method definition diagram. Initially the declaration contains the message selectors and formal parameters of the method. The user can add temporary variables required in the method by selecting the `add local variables` option in the *method declaration menu.* The process is similar to adding variables in a class definition view.

The lower subview of the method definition view is used to add statements in the method definition; the options `add message expression`, `add assignment` and `add return` are provided in the *method definition menu* for this purpose. For each of these options, a relevant pop-up menu is provided for specifying each component element of a statement. For example, to add a message expression, the user is first given a menu containing the different types of receiver of a message, i.e., variable, literal, class. When a type is selected, the user is prompted for the name (of the variable or class) or value (of the literal) and then the correct symbol for the receiver is added to the diagram automatically. The user is next prompted for the message, and the symbols for the message selectors and parameters are added to the diagram connected to the receiver. To insert a statement before another statement in the diagram, the user selects one of the insert options in the method definition menu: `insert message expression`, `insert assignment` and `insert return`. In each case, the user first indicates the statement in the diagram before which the new statement is to be added by selecting the top element of the statement (e.g., the receiver is the top element in a message expression), then proceeds as normal to specify the components of the new statement.

There is no restriction on the order in which the two sections of a method definition are completed, and because the two sections are viewed together, the user can edit each section taking into account what is present in the other. When a method definition is complete, the statements must be compiled into a form the interpreter can execute. The `accept` option in the method definition menu is used to compile the statements, rather like the `accept` menu option provided in the text view of the Smalltalk-80 browser for compiling a method. Sections 6.3 and 6.5 describe respectively the data structures used by the compilation process, and the behaviour of the interpreter and its instruction set used to express the result of the compilation process.

6.3 Data Structure Representation of a Specification

The specification of a system described by a set of specification diagrams is represented by data structures constructed as the diagrams are created using the editor. A brief overview of the data structures used is now provided.

A specification needs to contain the descriptions of system objects which define the top-level structure of a system in the specification's object interaction diagram. Each system object description includes the object's parent-child hierarchy and its life history. The parent-child hierarchy must maintain a collection of descriptions of the child objects in the object's state; each description of a child object must again describe the parent-child hierarchy and life history of the child object. This pattern repeats until a child object does not need to be defined further. A specification is implemented as an instance of the class `Specification` and descriptions of system objects and child objects are defined by the class `SystemObject`, which contains the instance variables `childObjects` and `lifeCycle`. Figure 6.4 shows the hierarchical nature of the data structures of a specification comprising a `Specification` object which refers to a set of `SystemObject` instances in its instance variable `systemObjects`; these latter objects describe the system objects in the specification's object interaction diagram. Each of the `SystemObject` instances in turn refers to other `SystemObject` instances which describe child objects of a system object. The life history of an object is described in detail in Section 6.4.

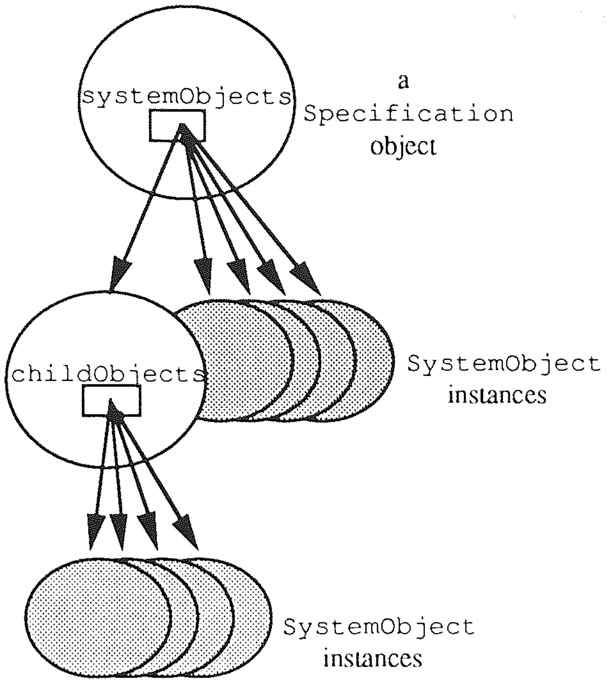


Figure 6.4 Hierarchical data structures of a specification.

A specification also contains a set of class definitions which implement objects in the specification. Each class definition stores details defined in the IVP and the CVP diagrams. The class `ClassDefinition` implements the class definitions. A class definition must contain a method for each message in the class's instance and class protocols. The method associated with a message is realised by the class `MethodDetail`. Apart from listing the local variables and formal parameters required in a method, a `MethodDetail` object contains the statements of the method. The three types of statement in a method are implemented by the classes `MessageExpression`, `Assignment` and `ReturnObject`.

Compilation of Methods

When the `accept` option is selected in a method definition view, the editor compiles the statements of the method in the view and the instructions generated from the compilation are also located in the method's `MethodDetail` definition. During the compilation of a method, some context sensitive checks are incorporated to ensure that, as far as possible, no illegal message send or access to a variable is accepted. In the case of a message send, a limit to the extent to which checking may be performed is imposed by the effect of dynamic binding — the binding of a method to a message occurs only during run-time so context sensitive checks on a message send cannot take place during compilation. However, it is possible to validate messages sent to the receiver, referred to by the pseudo-variables `self` and `super`; as explained in Section 5.5, `self` and `super` always refer to the message receiver in a method and their values cannot be altered dynamically within the method by an assignment during run time. It is therefore feasible to check that only instance messages defined in the receiver's class are sent to the receiver. Validation may also be carried out during compilation on messages sent to a class to ensure that the message is present in the class protocol of the class's definition.

During compilation, each statement in a method is compiled into a series of instructions in the interpreter's instruction set; each instruction is stored in the form of an `Array` object, containing the label (or name) of the instruction — which is a `Symbol` object — followed by any parameters the instruction requires. For example, one instruction in the instruction set is the unconditional jump instruction which requires one integer parameter; an example of this jump instruction is shown below using the Smalltalk-80 literal expression for an `Array` object:

#(jump 1)

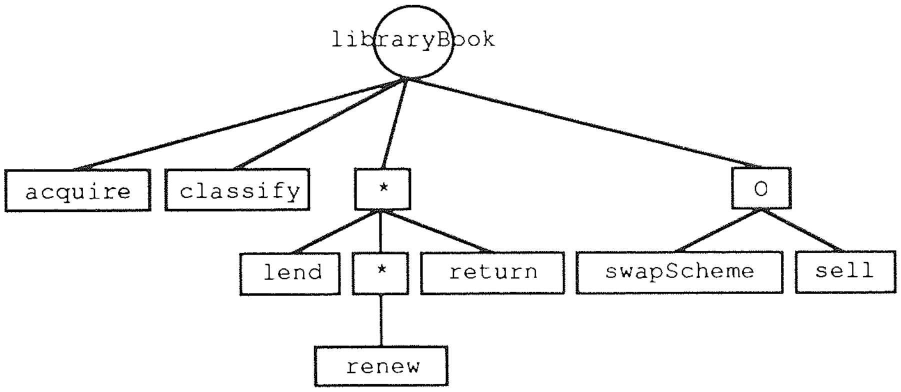
The interpreter's instruction set is described in more detail in Section 6.5.

6.4 Object Life Histories

Precede-sets

Each message in an object's life history diagram defines a *precede-set* which is a set of acceptable messages from which one must be sent to the object immediately before the message itself. For example, the life history diagram of the object `libraryBook` in Figure 6.5 shows that the message `lend` can only follow the `classify` or return message. The precede-set of `lend` therefore contains the messages `classify` and `return`. The precede-sets of all messages in an object's life history are stored in the instance variable `messages` in the object's `SystemObject` implementation . When an object receives a message, the interpreter first determines if the message has a precede-set, i.e., if the message is constrained by the object's life history. If the message has a precede-set, the latter is used to ascertain that the life history of the object will not be violated by the message send. The message's precede-set is searched to check if it includes the last message the object received; if the last message is present in the precede-set then the message receive is allowed to proceed. For example, assume the object `libraryBook` has received the message `acquire`. A following message send involving the message `lend` is invalid because the precede-set of `lend` does not contain the message `acquire`, and so the message receive will not be allowed to proceed.

A message's precede-set is constructed as the message is introduced into a life history diagram. Since a message may occur more than once in a life history diagram, the precede-set of the message needs to be updated each time the message is added in the diagram. To facilitate the process of constructing and updating the precede-sets of messages, sequences, iterations and selections in a life history are also implemented with their own precede-sets. Each of these precede-sets contains the possible messages that may be sent to an object immediately before a sequence, iteration or selection. The main structure of an object's life history is implemented as a sequence. The object should not receive any message before its life history sequence starts. The precede-set of a sequence is initialised to contain the `nil` object to indicate the absence of any message. For example, the sequence representing the life history of `libraryBook` has the precede-set `IdentitySet(nil)`.



Message	Precede-set
acquire	IdentitySet(nil)
classify	IdentitySet(acquire)
lend	IdentitySet(classify, return)
renew	IdentitySet(lend, renew)
return	IdentitySet(renew, lend)
swapScheme	IdentitySet(return, classify)
sell	IdentitySet(return, classify)

Sequence/Subtree	Precede-set
sequence	IdentitySet(nil)
iteration(lend, *, return)	IdentitySet(classify)
iteration(renew)	IdentitySet(lend)
selection(swapScheme, sell)	IdentitySet(return, classify)

Figure 6.5 Life history of the object `libraryBook` and the associated precede-sets.

The precede-set of an iteration or a selection depends on the position in a life history that the iteration or selection occupies. When an iteration or a selection is the first entry in a life history, its precede-set is initialised to the precede-set of the life history (i.e., a set containing the nil object). This happens because an object should not receive any message before the first entry in its life history. For an iteration or a selection occurring in any other position, its precede-set depends on the immediately preceding entry in the life history. If the preceding entry was a message, the precede-set of the iteration or selection must contain this message. For example, the iteration `(lend, *, return)` in `libraryBook`'s life history has the precede-set `IdentitySet(classify)`. But if the preceding entry was another iteration or selection, all the messages that could occur last in the preceding iteration or selection must be included in the precede-set of the iteration or selection itself. Figure 6.5 shows all the precede-sets of all entries in the libraryBook object life history diagram.

Last-sets

To facilitate the process of deriving precede-sets, each entry type (i.e., message, iteration and selection) defines a set of message(s) known as a *last-set*. The last-set of an entry contains the possible messages an object could have received last after the entry in the object's life history has occurred. The precede-set of an entry would be equivalent to the last-set of its preceding entry. The last-set of a message entry logically contains the message itself, e.g., the message entry classify in Figure 6.5 has the last-set IdentitySet(classify). All entries in a selection (including messages and iterations) have equal likelihood of occurring, so the selection's last-set must include the last-set of every entry it contains. For example, the last-set of the selection (swapScheme, sell) in Figure 6.5 contains the messages swapScheme and sell. The last-set of an iteration contains the last-set of the final entry in the iteration, but must also include the iteration's precede-set because the iteration might not occur, in which case the messages in the precede-set of the iteration become the last possible messages an object could have received after skipping the iteration. For example, the last-set of the iteration (lend,*,return) in Figure 6.5 contains the messages return and classify.

The precede-set of a message is also derived using the same principle as for iterations and selections, i.e., its precede-set is dependent on the position in a life history or a subtree in which the message is added. When the message is added as the first entry of a sequence, or an iteration or selection subtree, its precede-set would be equivalent to the precede-set of the sequence or subtree. When added in any other position, the message's precede-set is equivalent to the last-set of its preceding entry in the sequence or subtree. For example, in Figure 6.5 the precede-set of the message `return` is equivalent to the last-set of the iteration `(renew)`.

First-sets

Extra attention must be given to the precede-set(s) of the message(s) in the first entry of an iteration. After the last entry in the iteration has occurred, the whole iteration may repeat and the first entry in the iteration would occur again. The precede-set of a message first entry in an iteration must be augmented to include the last-set of the iteration. When the first entry in the iteration is a selection or another iteration, it is the precede-sets of the first messages in these subtrees that need to be expanded.

The set of message(s) in an iteration whose precede-sets need to be modified as described is called its *first-set*. To facilitate the derivation of an iteration's first-set, each message and selection entry also defines its own first-set. The first-set of a message contains only the message itself. The first-set of a selection would include the first-set of every entry in the selection, which may be a message or an iteration, since each entry in the selection can. The first-set of an iteration is therefore equivalent to the first-set of the iteration's first entry, which could be a message, a selection or another iteration. For example, the first-set of the iteration `(lend, *, return)` in Figure 6.5 contains the message `lend`; the precede-set of `lend` contains `classify`, but must also be expanded to include the last-set of the iteration which contains the message `return`.

Implementation

The main building blocks of life histories are implemented by the classes `Sequence`, `Iteration` and `Selection`. Each class defines an instance variable `precedeSet`. Messages in a life history are implemented by the class `MessageSymbol`; as described in the section *Precede-sets*, however, the precede-sets of messages are stored in the `SystemObject` instance of the object which owns the life history.

The last-set and first-set of each entry type are not stored but are implemented as the instance messages `lastSet` and `first` of the classes `Iteration`, `Selection` and `MessageSymbol`. The last-set and first-set of an entry are used only when a life history is constructed but are not required during run-time, so there is no advantage in storing these information for each entry. Figures 6.6 and 6.7 show the implementation of these instance messages in the three classes.

```
lastSet
|idSet|
idSet <- IdentitySet new.
idSet add: messageName.
^idSet
```

(a)

```
lastSet
|idSet|
idSet <- IdentitySet new.
self do: [:event| event lastSet
                   do: [:item| idSet add: item]].
^idSet
```

(b)

```
lastSet
|idSet|
idSet <- (self at: self size) lastSet.
self precedeSet do: [:item|idSet add: item].
^idSet
```

(c)

Figure 6.6 Instance method `lastSet` of the classes (a) `MessageSymbol`, (b) `Selection` and (c) `Iteration`.

```
first
^(self at: 1) first
```

(a)

```
first
|idSet|
idSet <- IdentitySet new.
idSet add: messageName.
^idSet
```

(b)

```
first
|idSet|
idSet <- IdentitySet new.
self do: [:event| event first do:
        [:item| idSet add: item]].
^idSet
```

(c)

Figure 6.7 Instance method `first` of the classes (a) `Iteration`, (b) `MessageSymbol` and (c) `Selection`.

6.5 The Interpreter

Implementation and Behaviour of the Interpreter

The interpreter executes the compiled instructions of a method. The interpreter is implemented by the class `Interpreter` and is referred to by the Smalltalk-80 global variable `SpecificationInterpreter`. The details of the required implementation for the interpreter are derived from understanding the virtual machine of the Smalltalk-80 system. Five items of information are required by the interpreter for executing the instructions of a method:

i. the instructions to be executed

ii. the location of the next instruction to be executed — indicated by an instruction pointer

iii. the receiver and arguments of the message that invoked the method

iv. any local variables required by the method

v. a stack.

The execution of most instructions (described in the following section Instruction Set) involves the stack. A push instruction causes an object to be located and added to the stack. A store instruction results in an object on the top of the stack being removed and placed in a variable. A send instruction removes the receiver and arguments of a message from the stack, and when the result of the message send is computed, it is pushed onto the stack. The interpreter repeatedly performs a three-step cycle when executing a method as follows:

i. fetch the next instruction from the method designated by the instruction pointer

ii. increment the instruction pointer

iii. perform the function specified by the instruction.

Push, store, and jump instructions require only small changes to the information held by the interpreter — objects may be added to or removed from the stack, and the instruction pointer is modified after every instruction. However, send and return instructions may require greater changes to the interpreter's state. When a message send is encountered, information used by the interpreter has to be changed in order to execute the instructions of the method corresponding to the message send. The information currently used by the interpreter must be remembered before the send instruction is executed because the execution of the remaining instructions in the method following the send instruction must be resumed after the value of the message send is returned. The logical way to implement this is to maintain the five-part information relating to the execution of a method in an object called a *context*. Whenever the interpreter arrives at a send instruction, a new context is created for the message send to hold the information associated with the method of the message. The interpreter, therefore, must be able to deal with more than one context at any one time.

The context containing the information currently being used by the interpreter is called the *active context*. When a new context is created in response to a send instruction, the active context becomes suspended and the new context succeeds as the active context. The order of creation of contexts must be maintained so that when the interpreter finishes executing the instructions of an active context, the suspended context which gave rise to the active context may be reactivated and its execution resumed from where it was previously halted. A stack, called a *context stack*, is used for storing contexts in the correct order in the interpreter.

The execution of a block's instructions may be compared with executing the instructions of a method. The `#evaluateBlock` instruction is therefore implemented in a similar way to a send instruction, involving a new context to be created for the block's execution. The meaning of a block is dependent on the method in which the block resides; the block shares the same receiver, message arguments, and local variables of its method. Hence, the receiver, arguments and local variables of the block's context refer to the same values as the suspended context of the block's parent method. On the other hand, the block's context contains its own set of instructions compiled from the statements in the block, and must contain its own instruction pointer and stack.

When an object receives a message and a method is executed, the interpreter is required to reflect the effect of the execution on the life history of the object in its life history diagram, and also show the messages the object sends to its child and colleague objects in the object's attribute diagram. This animation of behaviour by the interpreter is described in more detail in a later section (*System Execution and Behaviour Animation*). The interpreter has to access the object's detail in its `SystemObject` definition in order to carry out this animation.

A child or colleague object receiving a message may also require the same animation of its behaviour. The interpreter will also need to access the `SystemObject` definition of the child or colleague object. When the child or colleague object finishes its execution, the interpreter may need to return to the definition of the parent object to continue animation of the object's method execution. The interpreter therefore needs to be able to refer to the `SystemObject` definition of more than one object at any time. This is implemented by another stack in the interpreter. The `SystemObject` instance of a child or colleague object is added to or removed from this stack when the object receives a message or completes its execution.

Instruction Set

The instruction set of the interpreter is based on the instruction set used by the virtual machine of the Smalltalk-80 system. The interpreter's instructions may be grouped into six main categories: push, store, send, return, jump, and evaluate. The push instructions are used to indicate the source of an object to be added onto the top of the interpreter's stack. Possible sources include:

- a receiver
- a class
- a literal value
- instance variables of a receiver
- class variables of a receiver's class
- a block.

The store instructions are used in the compilation of `Assignment` objects. Instructions from compiling an `Assignment` instance first compute the value to be assigned, and an appropriate push instruction is used to place the value on the top of the interpreter's stack. A store instruction is then used to move the stack-top object into the variable specified by the store instruction. The variables whose values may be updated by a store instruction include:

- instance variables of a receiver
- local variables of a method
- class variables of a receiver's class.

In compiling a MessageExpression object, a send instruction is used to specify the selector of a message and the number of arguments the message expects in the message expression. The receiver and arguments of the message must be found on the interpreter's stack top before the execution of a send instruction. The appropriate push instructions are therefore generated during the compilation before the send instruction. The execution of a send instruction removes the message's receiver and arguments from the stack and replaces them with the value of the message send.

The return instruction #returnStackTop is generated in compiling a ReturnObject instance, and is used to indicate to the interpreter that the execution of a method's instructions is complete and to return the value of the message send which invoked the method. The value to be returned is removed from the top of the interpreter's stack; therefore, an appropriate push instruction is included by the compilation process before #returnStackTop. If the value to be returned happens to be a message expression or an assignment, the correct instructions to compute the return value are generated by the compilation before the push instruction.

Normally, the interpreter executes instructions sequentially in the order they were generated during the method's compilation. A jump instruction, however, is used to alter the interpreter's execution sequence. There are two types of jump instruction: unconditional and conditional. The unconditional jump instruction transfers control whenever it is encountered, while the conditional jump instruction will only transfer control if the interpreter's stack top is an expected value — a Boolean object. Both types of jump instruction specify the position of the next instruction to be executed in the form of an Integer offset relative to the location of the jump instruction itself; the offset indicates to the interpreter the number of instructions to skip following the jump instruction. Jump instructions are used to optimise the implementation of control structures provided by `Boolean` objects and blocks (see Section 5.5).

The `#evaluateBlock` instruction implements the message `value` sent to a block object. `#evaluateBlock` starts the execution of the instructions of a compiled block located at the top of the interpreter's stack; the compilation of a block object therefore generates an appropriate push instruction before `#evaluateBlock` to place the compiled instructions of the block on the stack. Figure 6.8 shows the compiled instructions of the instance method `push: item` (Figure 5.20) defined in the example class `Stack` in Figure 5.8; the compiled instructions are shown using the Smalltalk-80 literal expressions for representing `OrderedCollection` and `Array` objects.

```
OrderedCollection (
(pushIntvar maxStLength)
(receiver maxStLength)
(pushIntVar stLength)
(send an InteractionMessage 1)
(receiver nil)
(storeTempVar stackFull)
(pushTempVar stackFull)
(pushTempVar stackFull)
(jumpIfTrue 4)
(pushBlock OrderedCollection (
              (pushIntVar stLength)
              (receiver stLength)
              (pushLit 1)
              (send an InteractionMessage 1)
              (receiver nil)
              (storeIntVar stLength)
              (pushIntVar stSpace)
              (receiver stSpace)
              (pushIntVar stLength)
              (pushArg item)
              (send an InteractionMessage 2)
              (reciever nil)
              (returnStackTop)))
(blockReceiver)
(evaluateBlock (236 @ 150 corner: 297 @ 186))
(jump 1)
(pushLit nil)
(receiver nil)
(pushReceiver)
(pushLit full)
(send an InteractionMessage 1)
(receiver nil))
```

Figure 6.8 Instructions generated by compiling the method `push: item`.

Some of the instructions in Figure 6.8 which have not been described, including `#receiver` and `#blockReceiver`, are used by the interpreter to facilitate its animation process during the execution of a system. Figure 6.9 shows the data structure representation of the statements and compiled instructions in the `MethodDetail` instance of the method `push: item`.

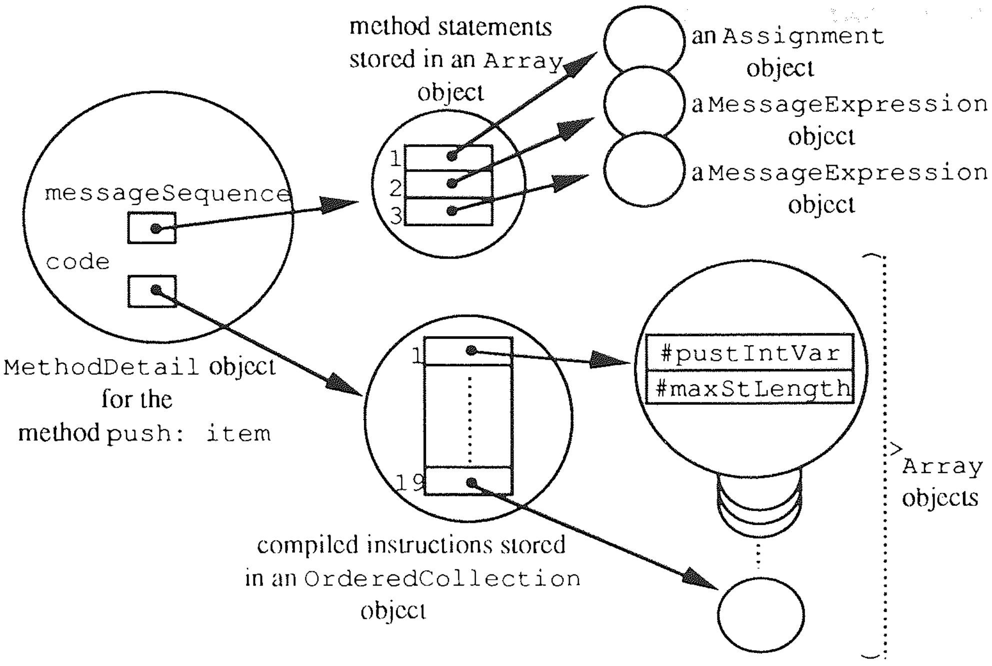

method statements stored in an Array object
an Assignment object
aMessageExpression object
aMessageExpression object
1
2
3
messageSequence
code
MethodDetail object for the method push: item
1
19
#pustIntVar
#maxStLength
Array objects
compiled instructions stored in an OrderedCollection object

Figure 6.9 Data structure representation of the statements and compiled instructions of the method push: item.

Object Instantiations During System Execution

The tool allows classes in the Smalltalk-80 class library to be instantiated at run-time during the execution of a system. On the other hand, run-time objects "instantiated" from classes defined using the tool as part of the specification are realised by the class InstanceObject. An InstanceObject instance represents an object and contains its own copy of all the instance variables named in the IVP diagram of the object's class. The instance variables refer to actual state values of the object, and these state values will in turn be instances of InstanceObject, or instances of Smalltalk-80 classes. Each InstanceObject instance representing an object contains the name of the object's class to facilitate access to the instance method definitions in the class. Each InstanceObject instance also has to remember the latest message it has received; this information is used by the interpreter for checking that the life history of the object is not violated. The information held by the class InstanceObject is found in the instance variables instanceVariables, class and life. For example, Figure 6.10

shows the instantiation of the class `Stack` defined in Figure 5.8. Each instantiation creates an `InstanceObject` instance representing a `Stack` object.

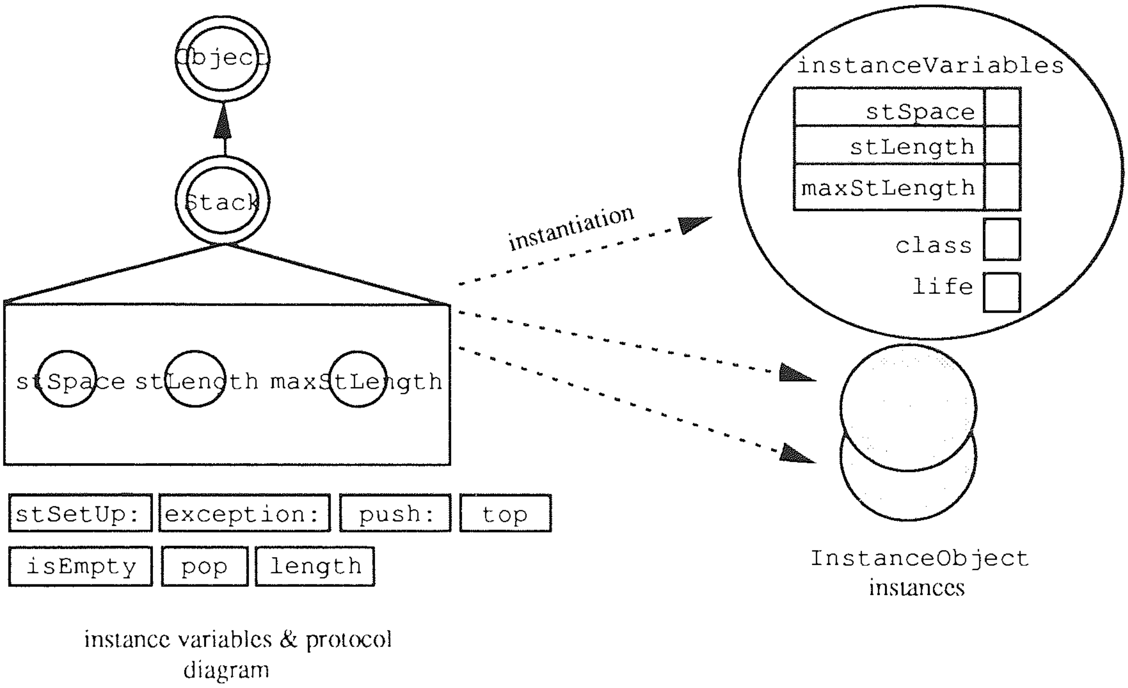


Figure 6.10 Instantiations of the class `Stack`.

Execution of Message Sends

A send instruction specifies the message selector of a message send and the number of parameters the message requires. When a send instruction is encountered, the interpreter first removes the correct number of parameters and the receiver of the message from the stack of the active context. If the receiver has been instantiated from a Smalltalk-80 pre-defined class (e.g., the receiver may be an `Array` object), the interpreter effects the message send by sending the receiver the instance message `perform:withArguments:`, with the message selector and the parameter objects as the arguments. `perform:withArguments:` is implemented by the Smalltalk-80 class `Object` — `Object` is the root class in the Smalltalk-80 class hierarchy and so its protocol is inherited by all classes in the hierarchy. `perform:withArguments:` returns the result of the message send to the receiver. Therefore, for a message receiver instantiated from a Smalltalk-80 class, the interpreter makes use of the Smalltalk-80 virtual machine to carry out its message sending function, thereby avoiding the need to create a new context for the execution of the message send. On the other hand, if the receiver of the message send was an `InstanceObject` object (i.e., the receiver had been instantiated from a class defined using the editor), the interpreter is required to perform the message send itself and a new context is therefore created and added to the context stack.

The method corresponding to the message selector specified in the send instruction must be located to obtain the compiled instructions the interpreter has to execute; this entails searching the class hierarchy of the specification. Once the method is located, the interpreter starts its three-step cycle to execute the instructions. The cycle of the interpreter is performed by its instance method `executeCode: code` (Figure 6.11):

```
executeCode: code
|activeContext value|
activeContext <- contextStack last.
activeContext instructionPointer: 1.
[activeContext instructionPointer <= code size]
  whileTrue: [value <- self execute:
                         (code at: context instructionPointer).
              (value isMemberOf: Interpreter) not
                 ifTrue: [^value].
               activeContext instructionPointer:
                        activeContext instructionPointer + 1]
```

Figure 6.11 Interpreter's instance method for executing a method.

The formal parameter `code` contains the instructions of the method to be executed. The instruction pointer of the active context is initialised and the interpreter starts the execution of the method at the current instruction indicated by the instruction pointer. The message `execute:` executes each instruction and assigns the result to the variable `value`. As explained in the previous section (*Instruction Set*), the purpose of the `#returnStackTop` instruction is to terminate execution of a method and return the value of a message send which invoked the method. The result of executing the

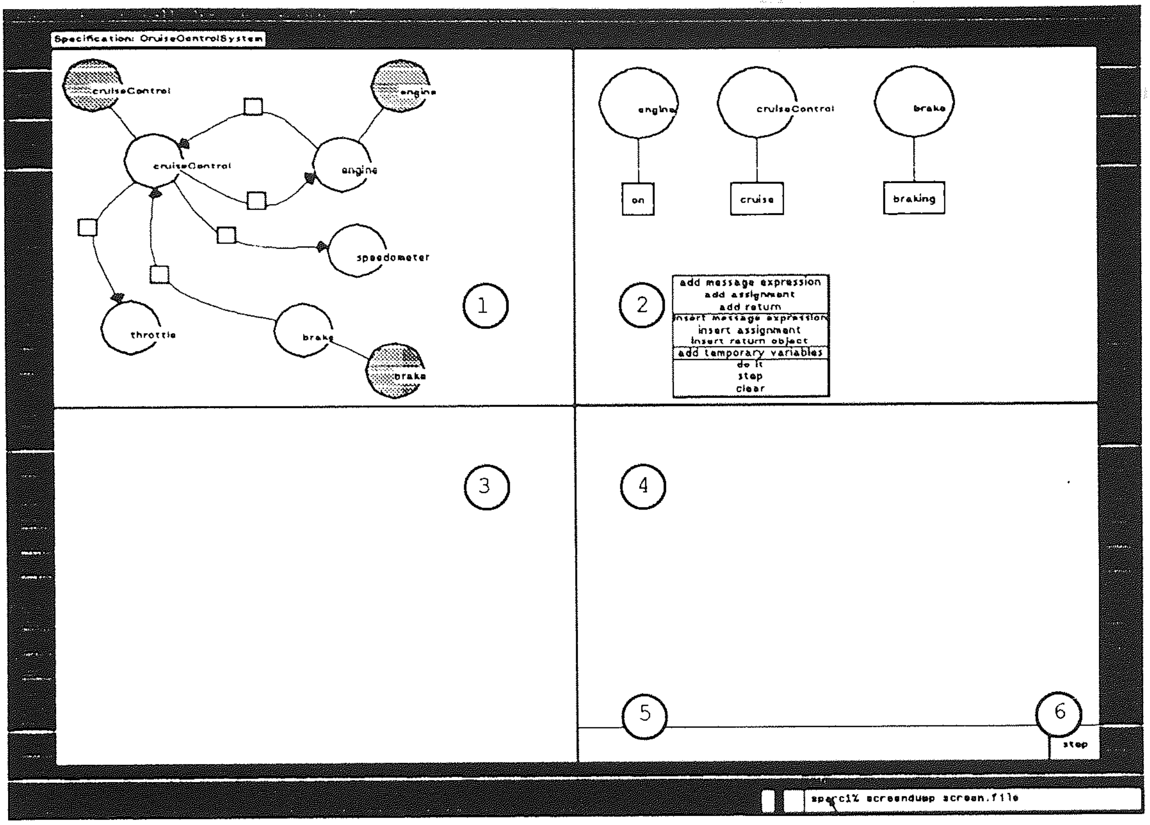
#returnStackTop instruction causes the method execute: to return the value of the message send which is assigned to value. When an instruction other than #returnStackTop is executed, the interpreter itself is returned by execute: and assigned to value. This is the effect of the Smalltalk-80 system's message sending function which returns the receiver of a message as the default value in the absence of a return expression in the message's method — in this case, the receiver of execute: is the interpreter itself. Hence, by checking the class of an object referred to by the value after each instruction's execution, the interpreter can detect when the #returnStackTop instruction is executed. When this occurs, the interpreter's execution of the method is stopped and value is returned from executeCode: as the result of the message send. Otherwise, execution of the method's instructions continues and so the instruction pointer is incremented.

When execute: code completes the execution of the method, the active context of the message send is removed from the interpreter's context stack. The execution of the method must return the value of the message send to the current active context; this value is placed at the top of the stack in the context. If the value returned was the interpreter, it is clear that the method of the message send does not contain a return expression; the receiver of the message send is regarded as the latter's result and is placed on the stack, replicating the default behaviour of the Smalltalk-80 system. If the value returned was not the interpreter, the value is placed on the stack in the active context.

System Execution and System Behaviour Animation

Along with executing the activities of a system described in a specification, an important function of the interpreter must be to animate this execution so as to provide some visual feedback to demonstrate the dynamic behaviour of the system during execution. A system is executed by sending messages to system objects in the object interaction diagram of the system's specification. As explained in Section 5.3, certain system objects of a system are designated as interface objects of the system and these interface objects can react to messages sent to the system from the system's environment (e.g., the user); interface system objects are identified by their connections to their own external entities in the object interaction diagram. A *system execution diagram* is used to specify operations to be performed on a system in terms of message sends to interface system objects. A new system execution diagram is displayed in a *specification execution view* in subview ② in Figure 6.12 when the `execute` option is selected in the specification menu (Figure 6.2). A system execution diagram is constructed with the same editing functions as for a method definition diagram, as can be shown by comparing the method definition menu and the *system execution menu* in Figures 6.2 and 6.12 respectively.

When the system execution diagram is completed, the operations specified in the diagram may be executed by selecting the `do it` or `step` options in the diagram's menu. Both options start the execution of the system but in different modes. The `step` mode pauses the system execution after each message send, and resumes execution when the step button (subview ⑥ in Figure 6.12) is pressed; the `step` mode thus allows the user to observe the execution at a preferred pace. The `do it` mode enables the execution of the system to be animated at a constant pace without interruption.



```
sparc1% screendump screen.file
```

Figure 6.12 Specification execution view.

When the execution of the operations specified in a system execution diagram commences, the interpreter highlights the current message send being performed in subview ② as execution progresses.

A system object reacting to a message it receives may send messages to other system objects in the object interaction diagram to complete its task. Messages are thus propagated from object to object amongst the system objects in the object interaction diagram which together contribute to the behaviour of the system in response to a message. The messaging activities of the system objects during the operation of the system is shown dynamically by displaying the flow of messages as they are sent amongst the network of system objects. A message flow is shown by highlighting the message symbol of the message channel connecting the system objects in the object interaction diagram and annotating with the name of the message in subview ① in Figure 6.12.

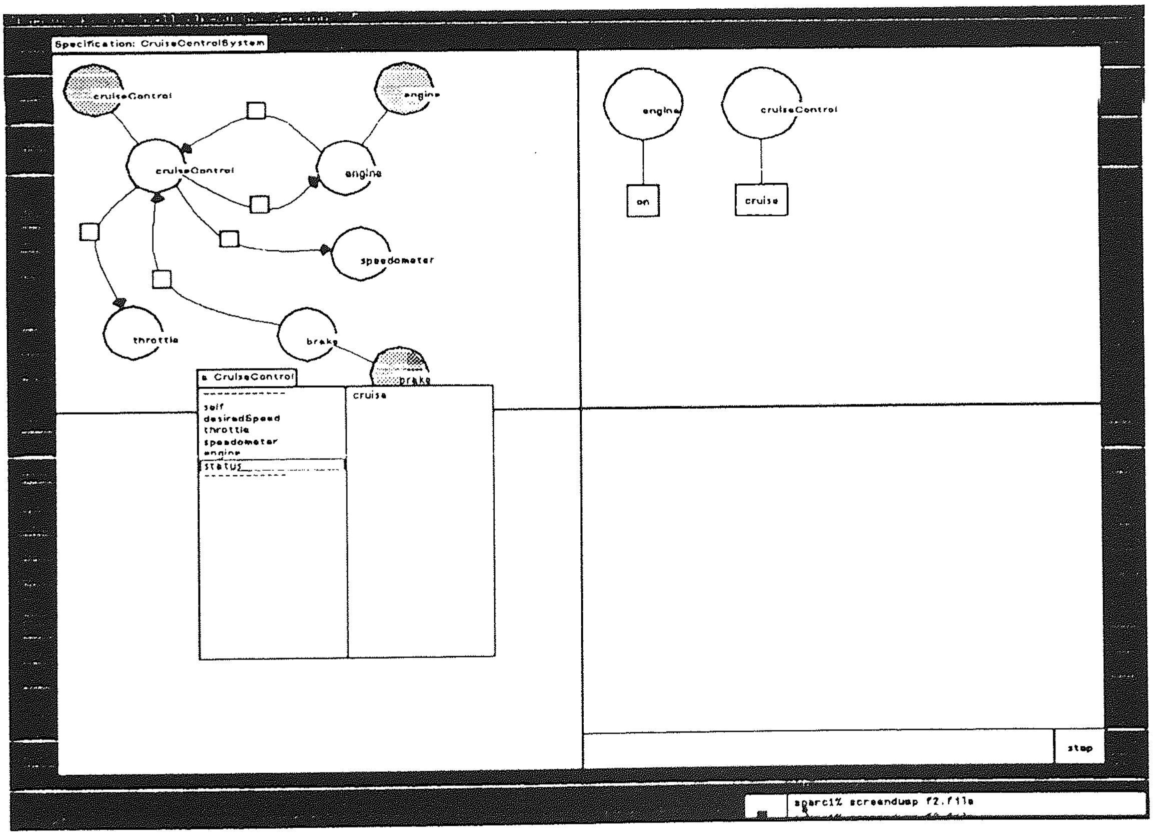
A message received by a system object may also cause the system object to send messages to its state objects to carry out the function of the message; this is shown using the parent-child hierarchy in the system object's object definition diagram, displaying each message flow to a child object dynamically as it occurs in subview ③ in Figure 6.12. Displaying the message communication from a parent to its child objects is not limited to the top-level system objects but is also used for lower-level objects when they send messages to their state objects.

As a message is being sent to an object in a system, the object's life history, if defined, is first examined to ensure that the receipt of the message will not violate the object's life history (see Section 6.4). If the message send to the object is permitted to proceed, it is useful to be able to observe the effect of receiving the message on the life history of the receiver. In this respect, the interpreter highlights the stage in the object's life history diagram the object has reached on receiving the message in subview ④ in Figure 6.12. The message subview ⑤ in Figure 6.12 displays the current message being sent in the system or an error message when an object's life history is violated.

Appendix A contains a series of snapshots showing the execution of the system in the example specification of an automobile cruise control system described in Section 5.6; these snapshots provide a flavour of the animation capability of the interpreter.

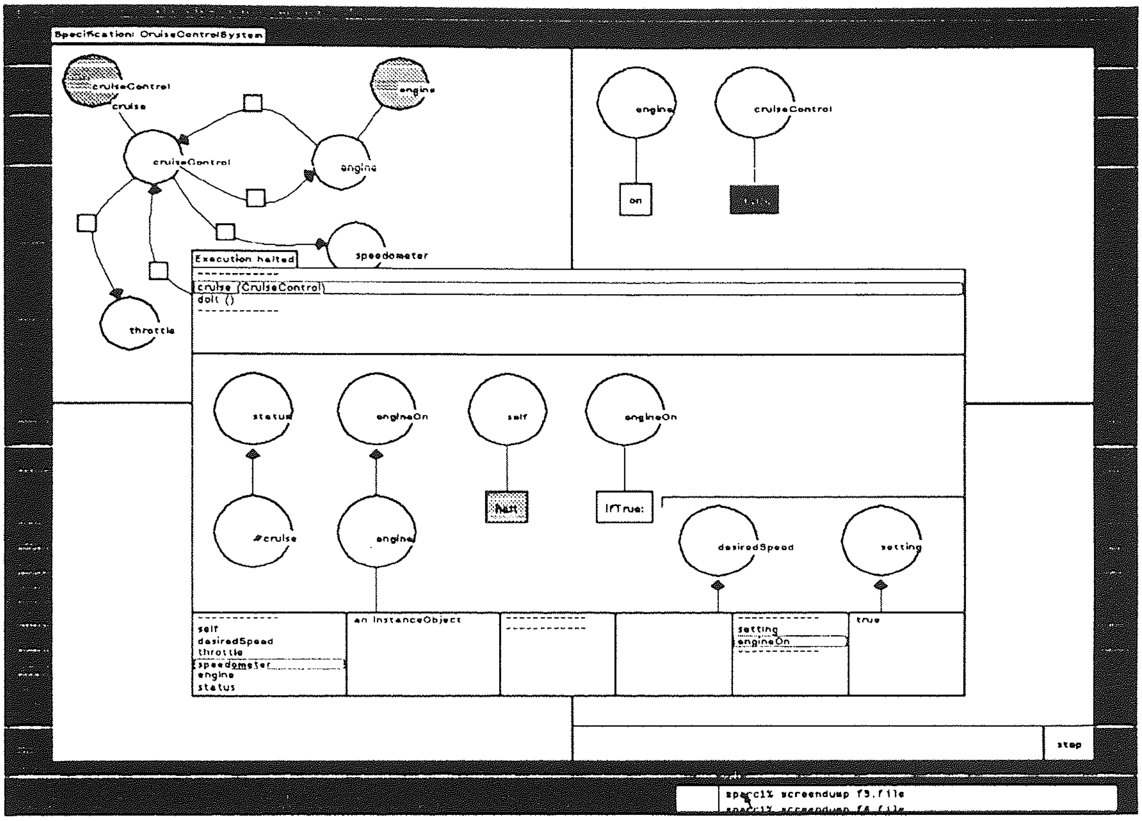
In the `step` mode of system execution, the user may examine the state values of system objects in the object interaction diagram during an execution pause by selecting the `inspect` option in the system object menu. An *inspector view* is opened listing the state objects present in the designated system object. The state of any state object may also be inspected by again selecting the `inspect` option in the *inspector* *view menu* when the state object is selected in the inspector view. Figure 6.13 shows an inspector view opened on the system object `cruiseControl`.

The interpreter also provides the facility whereby the user may choose to pause execution of the system at a designated point. The user may interrupt execution by using the message send `self halt` in either a method definition diagram, or the system execution diagram. In both cases, an *execution halt view* appears which allows the user to inspect the interpreter's context stack and the values of objects in each context. On closing the execution halt view, execution of the system resumes. Figure 6.14 shows the halting of system execution when the message send `self halt` is encountered while the system object `cruiseControl` is reacting to the message `cruise`.



```
sparc1% screendump f2.file
```

Figure 6.13 An inspector view on the object `cruiseControl`.



```
sparc1% screendump f3.file
```

Figure 6.14 A halt execution view.

Chapter 7 Conclusion

7.1 The Operational and Object-oriented Paradigms

The aim of this research has been to explore the feasibility of developing an operational specification technique specifically for object-oriented software development, and to implement a tool which can support the specification process. An important objective of the work has been that a specification developed using the technique is not only executable, but also lends itself to animation of its execution; the tool must therefore facilitate this objective. Analysis of some current object-oriented methods has shown that the development of a new graphical notation has been necessary which can represent both the executable semantics in an object-oriented system to enable the execution of a specification, as well as the system's dynamic behaviour so that the execution may be animated. Guidelines for using the notation have also been provided.

The tool implemented comprises an editor to facilitate the input of specifications, and an interpreter to execute and animate the execution of specifications. Executable semantics and dynamic information about an object-oriented system is specified using specification diagrams expressed in the notation. The editor used for specification input is able to build up a data structure representation which the interpreter uses to execute the specification. The interpreter animates this execution graphically based on the specification diagrams. Figure 7.1 shows the components of the work implemented.

The result of the research carried out suggests that the approach of incorporating executable semantics and dynamic behaviour in the specifications of object-oriented systems is possible, and the execution and animation of specifications by the tool is a useful facility for both the analyst and the user. Real-world entities in the problem domain of an object-oriented system are modelled in a specification in terms of objects and their behaviour. Functions of the system are then defined in terms of interactions amongst the objects. The tool's visualisation of the execution of a specification enables the behaviour of objects in the specified system to be demonstrated, as well as allowing the functionality of the system provided by the interactions of the objects to be revealed. The analyst's understanding of domain entities could be aided by using the tool, and a more accurate model of the problem domain can be created in the specification as a result. A system is likely to be more robust to future changes to its functionality if its implementation is based on a specification which describes the system's problem domain accurately. Through animation, the execution of a specification based on entities in the user's environment would be conceptually easier for the user to understand; the user can, therefore, help the analyst in validating the behaviour of objects in the specification and the functional requirements expressed.

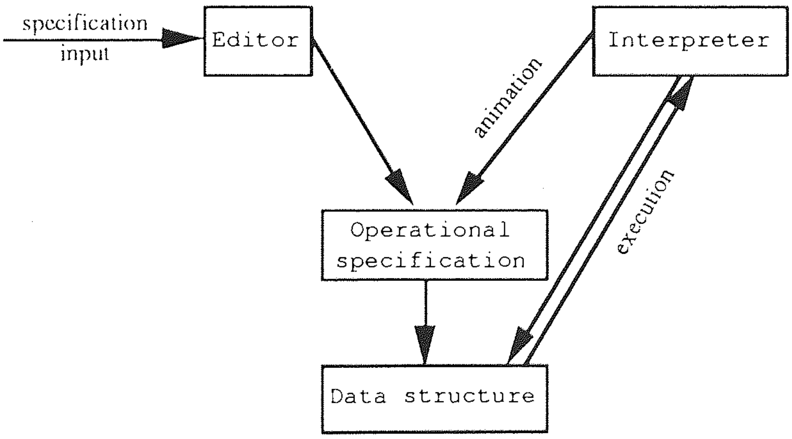


Figure 7.1 Components of the tool implemented.

The user may not often know fully what he requires or what the possible solutions are. It is useful for the developer and user to be able to modify the specification and experiment with different system architectures without incurring significant extra cost. The notation and tool together could therefore provide the potential for early validation during the specification phase which, as Agresti (1986a) notes, is difficult to achieve with traditional static natural-language specifications which do not possess executable semantics. The behaviour of a system specified in a static specification cannot be visible until the implementation phase; mistakes in the specification may then be costly and difficult to rectify in the code as explained in Section 4.1. An early validation capability could help to reduce development costs by lowering the probability of error in specifications.

Relationships Between the Two Paradigms

The emphasis of the operational approach is on the creation of executable specifications; this emphasis suggests that the dynamic characteristics of a system should be modelled in the system's specification. The initial task of this work therefore involved identifying the dynamic behaviours in an object-oriented system which must be represented in a specification which would render the specification executable.

Operations in an object-oriented system are carried out by methods of objects in the system. Executable semantics are therefore provided by method definitions. The obvious aspect of the dynamic behaviour in an object-oriented system is the interactions amongst objects which provide the functionality of a system, and each object's interactions with other objects when executing a method in response to a message. State changes in an object imply that another aspect of the dynamic behaviour is the life cycle of an object in terms of the messages it receives throughout its life span. A further dynamic aspect would involve the method resolution for a message received by an object at run-time. The inheritance mechanism causes a dynamic search for the message's method which can propagate up an inheritance structure of the object's class before the right method is invoked.

An important point to stress is that the central aim of this research has not been to develop a graphical programming language. However, in order to support execution, low-level implementation details relating to the object's method must be expressed in a specification. This aspect of a specification therefore resembles an object-oriented language. It therefore appears that the requirement for executable semantics necessarily imposes a certain degree of implementation detail in a specification. Swartout & Balzer (1982) argue that the explicit separation of specification and implementation in traditional development is unrealistic as both sets of development decisions are very much intertwined.

7.2 The Software Life Cycle and Future Development

The Object-oriented Software Life Cycle

The problems associated with the traditional waterfall model of the software life cycle have been described in Section 3.1. One such problem relates to the model's inadequate support for the natural iteration that exists particularly in object-oriented software development. The fountain and cluster models described in Section 3.2 have been proposed to remedy this problem. By incorporating operational specification animation into the object-oriented development process, the fountain and cluster models may be enhanced to emphasise the potential for early validation provided. Significant iteration occurs at the specification phase in which a specification may be continually executed and refined to arrive at a satisfactory statement about the functionality of the required system. Validation and testing are therefore no longer limited to the end of the life cycle as shown in the fountain and cluster models. Maintenance to correct errors in the implementation due to mistakes in the specification may be reduced. Generalisation and aggregation of classes in the models could also occur at this early stage with the benefit and ease of being able to experiment with different class structures and observe the behaviours of objects.

Future Development

The data structure representation of a specification may provide possible input to a code generator tool which would produce an initial implementation of a system in an object-oriented language. This approach would ensure consistency between the specification and its implementation. Different code generators may be used to derived implementations of the same system in different object-oriented languages. Currently, data structure representations of classes defined in a specification can be 'filed out' using the editor and then reused in other specifications by filing in. This reuse could also be extended to apply to other components of a specification such as object definitions and object interactions. The facility to incorporate pre-defined specification components could facilitate the reuse of specifications as envisaged by Balzer *et al* (1983).

Single inheritance only has been considered in the notation and tool. The notation should not require substantial extension to allow representation of multiple inheritance amongst classes. The main addition required in the notation to incorporate multiple inheritance would involve a way to express conflict resolution rules for resolving ambiguities that can arise in multiple inheritance as described in Section 2.5. At present the dynamic resolution of method invocation at run-time attributed to the inheritance mechanism is presented in the notation implicitly through the class inheritance relationship. The animation of a specification's execution does not explicitly show the search for a method when an object receives a message. It would be useful for this feature to be present in the tool's animation capability such that it would be possible to observe the effect of the inheritance mechanism on the behaviour of objects. This feature would be even more valuable with the inclusion of multiple inheritance as the inheritance structure could become extremely complex and the conflict resolution would need to be evaluated.

Two limitations exist in the present form of the notation. Firstly, messages in an object's protocol which do not appear in the object's life history diagram (i.e., messages which are not constrained by the order they are sent to the object) are not recorded anywhere else in the notation. The notation should be extended so that any uncontrained message of an object can be documented. Secondly, an improvement to the life history diagram would be to make it possible to denote time ordering of messages an object sends itself within a method (i.e., `self` messages in Smalltalk or `this` messages in C++) . This would involve allowing a message node in a diagram to own a sequence subtree containing iterations and/or selections of `self` or `this` messages.

The user interface of the tool should also be improved, for example in the execution view. In the execution view, a static layout is used where subviews share the display area of the view. This arrangement is not very practical for large specification diagrams as during execution not all parts of a diagram in each subview may be visible. Although horizontal and vertical scroll bars are provided in each subview, execution in the `do it` mode suspends scrolling. A more flexible arrangement may be to allow the user to select which subviews to focus on and to position and size each subview independently before execution.

7.3 Summary

This research has identified the potential benefit of joining the operational and object-oriented paradigms to provide a specification technique which combines the prototyping capability of executable specifications with the concepts of data abstraction, information hiding and reuse. It has been shown that the union may be achieved by representing the dynamic behaviours of objects in an object-oriented system in a specification. Different categories of these dynamic behaviours have been identified. The tool implemented has also demonstrated that it is possible to animate the execution of a specification produced using the method to provide early validation of functional requirements traditionally available only by building prototypes. The concepts used help the developer manage the complexity of the problem domain, and also ensure that a specification closely reflects the problem domain so that the specification is easier for the user to understand. Validation may be done more effectively because the user can understand the content of the specification and its execution. A more accurate model of the problem-domain may be derived through the animation of the specification's execution; when the model is incorporated in the implementated system, it can help to ensure that the system is more robust to cope with future changes to its functionality.

Research effort in the object-oriented paradigm has so far concentrated on developing programming languages, and analysis and design methods for object-oriented software development. With the increasing amount of investment in object-oriented systems, this research has hopefully shown that further work on developing the object-oriented software life cycle is worthwhile and can improve the chances of producing software that meet users's requirements.

References

Abbott, R. J., (1983), "Program Design by Informal English Descriptions", *Communications of the ACM*, **26**(11), November, pp.882-894.

Agha, G., (1991), *Actors: A Model of Concurrent Computation*, Massachusetts: MIT Press.

Agresti, W. W., (1986), "Part I. Introduction", in *New Paradigms for Software Development*, (ed. W. W. Agresti), IEEE Computer Society Press, pp.1

Agresti, W. W., (1986), "What are the new Paradigms", in *New Paradigms for Software Development* , ed. W. W. Agresti, IEEE Computer Society Press, pp.6-10.

Agresti, W. W., (1986a), "What are the New Paradigms ?", in *New Paradigms for Software Development*, (ed. W. W. Agresti), IEEE Computer Society Press, pp.6-10.

Alabiso, A., (1988), "Transformation of Data Flow Analysis Models to Object Oriented Design", in *OOPSLA'88 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, San Diego, California, USA, 25-30 Sept., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Sept., **23**(11), pp.335-353.

Alford, M., (1985), "SREM at the Age of Eight; The Distributed Computing Design System", *IEEE Computer*, April, pp.36-46.

Anderson, B., (1990), review of (Shlaer & Mellor, 1988), *Object-oriented Programming and Systems (BCS Specialist Group) Newsletter*, Issue 11, pp.21.

Ashworth, C. & Goodland, M., (1990), *SSADM A Practical Approach*, New York: McGraw-Hill.

Atkins, M. C. & Brown, A. W., (1991), "Principles of Object-oriented Systems", in *Software Engineer's Reference Book*, (ed. J. A. McDermid), Jordan Hill, Oxford: Butterworth-Heinemann.

Atkinson, C., Goldsack, S., Di Maio, A. & Bayan, R., (1991), "Object-oriented Concurrency and Distribution in DRAGOON", *Journal of Object-oriented Programming*, **4**(1), March/April, pp.11-18.

Bailin, S. C., (1989), "An Object-oriented Requirements Specification Method", *Communications of the ACM*, May, **32**(5), pp.608-623.

Balzer, R. M., Goldman, N. M. & Wile, D. S., (1982), "Operational Specification as the Basis for Rapid Prototyping", *ACM SIGSOFT Software Engineering Notes*, **7**(5), Dec., pp.3-16.

Balzer, R., Cheatham Jr., T. E. & Green, C., (1983), "Software Technology in the 1990's: Using a New Paradigm", *Computer*, Nov., pp.39-45.

Bauer, F., (1969), in *Software Engineering: A report on a conference sponsored by the NATO Science Committee*, (eds. P. Naur & B. Randell), NATO.

Beck, K. & Cunningham, W., (1989), "A Laboratory for Teaching Object-oriented Thinking", in *OOPSLA'89 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, New Orleans, Louisiana, USA, 1-6 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, **24**(10), pp.1-6.

Birchenough, A. & Cameron, J. R., (1989), "JSD and Object-oriented Design", in *The Jackson Approach to Software Development*, (ed. J. R. Cameron), IEEE Computer Society Press, pp.293-303.

Blair, G. S., Gallagher, J. J. & Malik, J., (1989), "Genericity vs Inheritance vs Delegation vs Conformance ... (Towards a Unifying Understanding of Objects)", *internal report* , Dept. of Computer Science, University of Lancaster.

Bobrow, D., DeMichiel, L. Gabriel, R., Keene, S., Kiczales, G. & Moon, D., (1988), "Common Lisp Object System Specification", *SIGPLAN Notices*, vol. 23, X3J13 Document 88-002R.

Boehm, B. W., (1988), "A Spiral Model of Software Development and Enhancement", *IEEE Computer* , May, pp.61-72.

Booch, G., (1982), "Object-Oriented Design", *Ada Letters* , **1**(3), pp.64-76.

Booch, G., (1986), "Object-Oriented Development", *IEEE Transactions on Software Engineering* , **SE-12**(2), pp.211-221.

Booch, G., (1990), "On the Concepts of Object-Oriented Design", in *Modern Software Engineering: Foundations and Current Perspectives* , (eds. P. A. Ng & R. T. Yeh), New York: Van Nostrand Reinhold, pp.165-204.

Booch, G., (1991), *Object-oriented Design with Applications*, California: Benjamin Cummings.

Buhr, R., (1988), "Machine Charts for Visual Prototyping in System Design", *SCE Report 88-2*, Carleton University, Ottawa, Canada.

Cameron, J. R., (1986), "An Overview of JSD", *IEEE Transactions on Software Engineering* , **SE-12**(2), pp.222-240.

Cameron, J. R., (1988), "The Modelling Phase of JSD", *Information and Software Technology*, July/August, **30**(6), pp.373-383.

Cardelli, L. & Wegner, P., (1985), "On Understanding Types, Data Abstraction, and Polymorphism", *ACM Computing Survey*, **17**(4), Dec., pp.481.

Chen, P., (1976), "The Entity-Relationship Model: Towards a Unified View of Data", *ACM Transactions on Database Systems*, **1**(1), pp.9-36.

Coad, P. & Yourdon, E., (1990), *Object-oriented Analysis*, Englewood Cliffs, NJ: Prentice-Hall.

Coad, P. & Yourdon, E., (1991), *Object-oriented Analysis*, (2nd edition), Englewood Cliffs, NJ: Prentice-Hall.

Cohen, D., Swartout, W. & Balzer, R., (1982), "Using Symbolic Execution to Characterise Behavior", *ACM SIGSOFT Software Engineering Notes*, **7**(5), Dec., pp.25-32.

Cook, S., (1986), "Languages and Objected-oriented Programming", *Software Engineering Journal*, March, pp.73-88.

Cook, S., (1987), "Object-oriented Programming in the UK", *British Computer Society Bulletin*, December, pp.22-23.

Cox, B. & Hunt, B., (1985), "Objects, Icons, and Software-ICs", *BYTE*, August, pp.161-176.

Cox, B. J., (1984), "Message/Object Programming: An Evolutionary Change in Programming Technology", *IEEE Software*, **1**(1), Jan., pp.50-61.

Dahl, O. & Nygaard, K., (1966), "Simula: An Algol based Simulation Language", *Communications of the ACM*, Sept., **9**(9), pp.671-678.

Dahl, O., Dijkstra, E. W. & Hoare, C. A. R., (1972), *Structured Programming*, London: Academic Press.

de Champeaux, D. & Faure, P., (1992), "A Comparative Study of Object-oriented Analysis Methods", *Journal of Object-oriented Programming*, March/April, pp.21-33.

DeMarco, L. P., (1979), *Structured Analysis and System Specification*, Englewood Cliffs, NJ: Prentice-Hall.

DeMarco, T., (1979), *Structured Analysis and System Specification*, Englewood Cliffs, NJ: Prentice-Hall.

Deutsch, M., (1989), "Design Reuse and Frameworks in the Smalltalk-80 System", in *Software Reusability*, Vol.II, (eds. T. J. Biggerstaff & A. J. Perlis), ACM Press, pp.57-71.

Feather, M. S., (1982), "Mappings for Rapid Prototyping", *ACM SIGSOFT Software Engineering Notes*, **7**(5), Dec., pp.17-24.

Fikes, R. & Kehler, T., (1985), "The Role of Frame-based Representation in Reasoning", *Communications of the ACM*, **28**(9), pp.904-920.

Gannon, J., Hamlet, R. & Mills, H., (1987), "Theory of Modules", *IEEE Transactions on Software Engineering*, **SE-13**(7), pp.820.

Gindre, C. & Sada, F., (1989), "A Development in Eiffel: Design and Implementation of a Network Simulator", *Journal of Object-oriented Programming*, **2**(1), pp.27-33.

Gladden, G. R., (1982), "Stop the Life Cycle — I Want to Get Off", *ACM Software Engineering Notes*, **7**(2), pp.35-39.

Golderg, A. & Robson, D., (1983), *SMALLTALK-80: The Language And Its Implementation*, Reading, MA: Addison-Wesley.

Golderg, A., (1984), *SMALLTALK-80: The Interactive Programming Environment*, Reading, MA: Addison-Wesley.

Graham, I., (1990), "Object-Orientation for Beginners: Part 1" in *OOPS (BCS Specialist Group) Object-oriented Programming and Systems Newsletter*, Issue 11, pp.6-9.

Graham, I., (1991), *Object-oriented Methods*, Reading, MA: Addison-Wesley.

Guttag, J., (1977), "Abstract Data Types and the Development of Data Structures", *Communications of the ACM*, June, **20**(6), pp.396-404.

Hanks, P. (ed.), (1986),*The Collins English Dictionary*, London: Collins.

Harel, D., (1987), "Statecharts: A Visual Formalism for Complex Systems", *Scientific Computer Programming*, vol.8, pp.231-274.

Harland, D. M., (1984), *Polymorphic Programming Languages: Design and Implementation*, Ellis Horwood.

Harland, D. M., (1988), *Recusiv: Object-oriented Computer Architecture*, Ellis Horwood.

Hatley, D. & Pirbhai, I., (1988), *Strategies for Real-Time System Specification*, New York: Dorset House.

Helm, R., Holland, I. M. & Gangopadhyay, D., (1990), "Contracts: Specifying Behavioural Compositions in Object-oriented Systems", in *ECOOP/OOPSLA'90 Proceedings of the Fourth European Conference on Object-oriented Programming/Object-oriented Programming Systems, Languages and Applications Conference Proceedings/*, Ottawa, Canada, 21-25 Oct., (eds. J. L. Archibald & K. C. B. Yakemovic), Special issue of ACM SIGPLAN Notices, pp.15-17.

Henderson, P., (1986), "Functional Programming, Formal Specification and Rapid Prototyping", *IEEE Transactions on Software Engineering* , **SE-12**(2), pp.241-250.

Henderson-Sellers, B. & Edwards, J. M., (1990), "The Object-oriented Systems Life Cycle", *Communications of the ACM*, Sept., **33**(9), pp.142-159.

Hodgeson, R., (1990), "Finding, Building and Reusing Objects", in *Proceedings of Object Oriented Design*, Unicom Seminars, Uxbridge.

HOOD Working Group, (1989), *HOOD REFERENCE MANUAL*, Noordwijk, The Netherlands: European Space Agency.

HOOD Working Group, (1989), *HOOD USER MANUAL*, Noordwijk, The Netherlands: European Space Agency.

Ingalls, D. H. H., (1978), "The Smalltalk-76 Programming System Design and Implementation", in *Conference Proceedings of the Fifth Annual ACM Synposium on Principles of Programming Languages*, New York: ACM Press, pp.9-16.

Jackson, M., (1983), *System Development*, Englewood Cliffs, NJ: Prentice-Hall.

Johnson, R. E., (1988), "Designing Reusable Classes", *Journal of Object-Oriented Programming*, June/July, pp.22-35.

Johnson, R. E., (1988), "Designing Reusable Classes", *Journal of Object-oriented Programming*, June/July, pp.22-35.

Jones, C. B., (1990), *Systematic Software Development Using VDM*, Englewood Cliffs, NJ: Prentice-Hall.

Kay, A. C., (1969), *The Reactive Engine*, PhD Thesis, University of Utah.

Keene, S. E., (1989), *Object-oriented Programming in Common Lisp: A Programmer's Guide to CLOS*, Reading, MA: Addison-Wesley.

Kernighan, B. W. & Ritchie, D. M., (1978), *The C Programming Language*, Englewood Cliffs, NJ: Prentice-Hall.

Kersten, M. L. & Schippers, F. H., (1986), "Towards an Object-centred Database Language", in *Proceedings of the 1986 International Workshop on Object-oriented Database Systems*, Washington, DC, USA, IEEE Computer Society Press, **24**(10), pp.353-361.

Khoshafian, S. N. & Copeland, G. P., (1986), "Object Identity", in *OOPSLA'86 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, OR, USA, 29 Sept.-2 Oct.,. (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **21**(11), pp.104-111.

Koffman, E. B., (1988), *Problem Solving and Structured Programming in Modula-2*, Reading, MA: Addison-Wesley.

Korson, T. & McGregor, J. D., (1990), "Understanding Object-oriented: A Unifying Paradigm", *Communications of the ACM*, Sept., Sept., **33**(9), pp.40-60.

Krasner, G. E. & Pope, S. T., (1988), "A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80", *Journal of Object-Oriented Programming*, Aug. /Sept. , pp.26-49.

Lalonde, W., (1989), "Designing Families of Data Types Using Exemplars", *ACM Transactions on Programming Languages and Systems*, **2**(3), April, pp.214.

Lea, D., (1988), *User's Guide to GNU C++ Library*, Cambridge, MA: Free Software Foundation.

Ledbetter, L. & Cox, B., (1985), "Software-ICs", *BYTE*, June, pp.307-316.

Lewis, C., (1988), "The Realisation of JSD Specifications in Object-oriented environments", *internal report*, Dept. of Computer Science & Applied Mathematics, Aston University, February.

Lewis, C., (1991), *The Realisation of JSD Specifications in Object-oriented Languages*, PhD Thesis, Aston University.

Lieberherr, K. J. & Riel, A. J., (1988), "Demeter: A Case Study of Software Growth Through Parameterized Classes", *Journal of Object-oriented Programming*, **1**(3), Aug./Sept., pp.8-22.

Lieberherr, K. J., Holland, I. & Riel, A. J., (1988), "Object-oriented Programming: An Objective Sense of Style", in *OOPSLA'88 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, San Diego, California, USA, 25-30 Sept., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Sept., **23**(11), pp.323-334.

Lieherman, H., (1986), "Using Prototypical Objects to Implement Shared Behaviour in Object-oriented Systems", in *OOPSLA'86 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, OR, USA, 29 Sept.-2 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **21**(11), pp.214-223.

Lim, J. & Johnson, R., (1989), "The Heart of Object-oriented Concurrent Programming", *SIGPLAN Notices*, **24**(4), pp.165.

Lippman, S. B., (1989), *C++ Primer*, Reading, MA: Addison-Wesley.

Liskov, B. & Zilles, S., (1977), "An Introduction to Formal Specifications of Data Abstractions", in *Current Trends in Programming Methodology*, Vol. 1, (ed. R. T. Yeh), Englewood Cliffs, NJ: Prentice-Hall, pp.1-32.

Liskov, B., (1988), "Data Abstraction and Hierarchy", *SIGPLAN Notices*, May, **23**(5), pp.19.

Liskov, P. H. & Zilles, S. N., (1974), "Programming with Abstract Data Types", *ACM SIGPLAN Notices*, **9**(4), pp.50-59.

Madsen, O. L. & Moller-Pedersen, B., (1988), "What Object-oriented Programming may be — and What it does not have to be", in *ECOOP'88 Proceedings of the Second European Conference on Object-oriented Programming*, Oslo, Norway, 15-17 Aug., (eds. S. Gjessing & K. Nygaard), Lecture Notes in Computer Science, No.322, Springer-Verlagf, pp.1-20.

Maes, P., (1987), "Concepts and Experiments in Computational Reflection", in *OOPSLA'87 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Orlando, Florida, USA, 4-8 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **22**(12), pp.147-155.

Maier, D., Stein, J., Otis, A. & Purdy, A., (1986), "Development of an Object-oriented DBMS", in *OOPSLA'86 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, OR, USA, 29 Sept.-2 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **21**(11), pp.472-482.

Malhortra, A., Thomas, J. C., Carroll, M. J. & Miller, L., (1980), "Cognitive Processes in Design", *Journal of Man-Machine Studies*, vol. 12, pp.119-140.

Martin, J., (1985), *Fourth Generation Languages Vol. 1*, Englewood Cliffs, NJ: Prentice-Hall.

Masiero, P. C. & Germano, F. S. R., (1988), "JSD as an Object Oriented Design Method", *ACM SIGSOFT Software Engineering Notes*, **13**(3), pp.22-23.

McCracken, D. D. & Jackson, M. A., (1981), "A Minority Dissenting Position", in *Systems Analysis and Design — A Foundation for the 1980's*, pp.551-553.

McCracken, D. D. & Jackson, M. A., (1982), "Life Cycle Concept Considered Harmful", *ACM Software Engineering Notes*, **7**(2), pp.28-32.

Meyer, B. , (1987), "Programming as Contracting", *report TR-EI-12/CO*, Goleta, C. A.:Interactive Software Engineering.

Meyer, B., (1979), *Object-oriented Software Construction*, Englewood Cliffs, NJ: Prentice-Hall.

Meyer, B., (1986), "Genericity Versus Inheritance", in *OOPSLA'86 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, OR, USA, 29 Sept.-2 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., pp.391-405.

Meyer, B., (1987), "Reusability: The Case for Object-Oriented Design", *IEEE Software*, **4**(2), pp.50-63.

Meyer, B., (1987), "Reusability: The Case for Object-oriented Design", *IEEE Software*, March, **4**(2), pp.50-64.

Meyer, B., (1988), *Object-oriented Software Construction*, New York, NY: Prentice-Hall.

Meyer, B., (1989), "From Structured Programming to Object-oriented Design: The Road to Eiffel", *Structured Programming*, Vol.1, pp.19-39.

Meyer, B., (1989a), "The New Culture of Software Development: Reflections on the Practice of Object-oriented Design", in *TOOLS'89 Proceedings of the First International Conference on the Technology of Object-oriented Languages and Systems*, Paris, France, 13-15 Nov., pp.13-23.

Micallef, J., (1989), "Encapsulation, Reusability, and Extensibility in Object-oriented Programming Languages", *Journal of Object-oriented Programming*, **1**(1), April/May, pp.15.

Miller, G. A., (1956), "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information", in *The Psychological Review*, **63**(2), March, pp.81-97.

Minsky, M., (1975), "A Framework for Representing Knowledge", in *The Psychology of Computer Vision*, (ed. P. H. Winston), New York: McGraw-Hill.

Monarchi, D. E. & Puhr, G. I., (1992), "Analysis and Modeling in Software Development", *Communications of the ACM*, May, **35**(9), pp.32-47.

Moon, D. A., (1986), "Object-oriented Programming with Flavours", in *OOPSLA'86 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, OR, USA, 29 Sept.-2 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **21**(11).

Morrison, R., Brown, A. L., Carrick, R., Connor, R. C. H., Dearle, A. & Atkinson, M. P., (1987), "Polymorphism, Persistence and Software Re-use in a Strongly-types Object-oriented Environment", *Software Engineering Journal*, **2**(6), pp.199-204.

Mullery, G. P., (1979), "CORE - A Method For Controlled Requirement Specification", *Proceedings of Fourth International Conference on Software Engineering*, Sept., pp.126-135.

Parnas, D., (1972), "On the Criteria To Be Used in Decomposing Systems into Modules", *Communications of the ACM*, December, pp.1053-1058.

Pascoe, G. A., (1986), "Elements of Object-oriented Programming", *BYTE*, August, pp.139-144.

Peterson, J. L., (1981), *Petri Net Theory and the Modeling of Systems*, Englewood Cliffs, NJ: Prentice-Hall.

Pinson, L. J. & Wiener, R. S., (1988), *An Introduction to Object-Oriented Programming and Smalltalk,* Reading, MA: Addison-Wesley.

Pressman, R. S., (1992), *Software Engineering: A practitioner's approach*, 3rd ed., New York: McGraw-Hill.

Pun, W. W. Y. & Winder, R. L. , (1989), "A Design Method for Object-oriented Implementation", *research note RN/89/79*, University College London.

Pun, W. W. Y. & Winder, R. L. , (1989b), "Automating Class Hierarchy Graph Construction", *research note RN/89/23*, University College London.

Pun, W. W. Y. & Winder, R. L., (1988), "Towards a Design Method for Object-oriented Programming", *research note RN/88/1*, Dept. of Computer Science, University College London.

Pun, W. W. Y. & Winder, R. L., (1989a), "A Design Method For Object-Oriented Programming", in *ECOOP'89 Proceedings of the Third European Conference on Object-oriented Programming*, Nottingham, UK, 10-14 July, (ed. S. Cook), Cambridge University Press, pp.225-240.

Purchase, J. A. & Winder, R. L., (1990), "Message Pattern Specifications: A New Technique for Handling Errors in Parallel Object Oriented Systems", in *ECOOP/OOPSLA'90 Proceedings of the Fourth European Conference on Object-oriented Programming/Object-oriented Programming Systems, Languages and Applications Conference Proceedings/*, Ottawa, Canada, 21-25 Oct., (eds. J. L. Archibald & K. C. B. Yakemovic), Special issue of ACM SIGPLAN Notices, pp.15-17.

Quillian, R., (1968), "Semantic Memory", in *Semantic Information Processing*, (ed. M Minsky), Massachusetts: MIT Press.

Ratcliff, B., (1987), *Software Engineering: Principles and Methods*, Oxford: Blackwell.

Renold, A., (1988), "Jackson System Development for Real Time Systems", *Scientia Electrica*, **34**(2), pp.3-43.

Rentsch, T., (1982), "Object Oriented Programming", *SIGPLAN Notices*, **17**(9), pp.51-57.

Roberts, G. A., Winder, R. L. & Wei, M., (1988), "the solve object oriented programming system for parallel computers", *Research Note WP10 19* , University College London, London, October.

Robinson, P. , (1989), "Hierarchic Object Oriented Design — HOOD", OOPS'23 *seminar notes*, London, UK, OOPS (BCS Specialist Group) Object-oriented Programming and Systems.

Robson, D., (1981), "Object-oriented Software Systems", *BYTE*, August, pp.5-8.

Ross, D. T., (1976), "Structured Analysis (SA): A Language for Communicating Ideas", *IEEE Transactions on Software Engineering*, **SE-3**(1), pp.16-34.

Ross, D. T., (1985), "Applications and Extensions of SADT", *Computer*, **18**(4), pp.25-34.

Ross, D., Goodenough, J. & Irvine, C., (1975), "Software Engineering Process, Principles, and Goals", *IEEE Computer*, **8**(5), May, pp.17-27.

Sakkinen, M., (1988), Comments on "the law of Demeter" and C++, *SIGPLAN Notices*, **23**(12), pp.38.

Sakkinen, M., (1989), "Disciplined Inheritance", in *ECOOP'89 Proceedings of the Third European Conference on Object-oriented Programming*, Nottingham, UK, 10-14 July, (ed. S. Cook), Cambridge University Press, pp.39-56.

Sanden, B., (1989), "An Entity-life Modeling Approach to the Design of Concurrent Software", *Communications of the ACM*, **32**(3), pp.330-343.

Schmucker, K., (1988), *Object-oriented Programming for the Macintosh*, Englewood Cliffs, NJ: Hayden.

Seidewitz, E. & Stark, M., (1986), "Towards a general Object-oriented Software Development Methodology", *Proceedings of the First International Conference on Ada Programming Language Applications for the NASA Space Station*,pp.D.4.6.1-D.4.6.14.

Seidewitz, E. & Stark, M., (1986), *General Object-oriented Software Development, Software Engineering Laboratory Series, SEL-86-002*, August, Greenbelt, M. D.: NASA Goddard Space Flight Center, pp.43.

Shankar, K., (1984), "Data Design: Types, Structures, and Abstractions", in *Handbook of Software Engineering*, New York: Van Nostrand Reinhold, pp.253.

Shaw, M., (1984), "Abstraction Techniques in Modern Programming Languages", *IEEE Software*, **1**(4), Oct., pp.10.

Shilling, J. J. & Sweeney, P. F., (1989), "Three Steps to Views: Extending the Object-oriented Paradigm", in *OOPSLA'89 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, New Orleans, Louisiana, USA, 1-6 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **24**(10), pp.353-361.

Shlaer, S. & Mellor, S. J., (1988), *Object-oriented Analysis: Modeling the World in Data*, Englewood Cliffs, NJ: Prentice-Hall.

Sincovec, R. & Wiener, R., (1984), "Modular Software Construction and Object-oriented Design using Ada", *Journal of Pascal, Ada & Modula-2*, Mar./Apr., pp.30-36.

Snyder, A., (1986), "Encapsulation and Inheritance in Object-Oriented Programming Languages", in *OOPSLA'86 Object-oriented Programming*

Systems, Languages and Applications Conference Proceedings, Portland, OR, USA, 29 Sept.-2 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **21**(11), pp.38-45.

Sommerville, I., (1992), *Software Engineering*, 4th ed., Reading, MA: Addison-Wesley.

Stefik, M. & Bobrow, D. G., (1986), "Object-oriented Programming: Themes and Variations", *The AI Magazine*, **6**(4), Winter, pp.40-62.

Stein, J., (1988), "Object-oriented Programming and Database Design", *Dr Dobb's Journal of Software Tools for the Professional Programmer*, No. 137, March, pp.18.

Stein, L. A., (1987), "Delegation is Inheritance", in *OOPSLA'87 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Orlando, Florida, USA, 4-8 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **22**(12), pp.138-146.

Stevens, W. P., Myer, G. J. & Constantine, L. L., (1974), "Structured Design", *IBM Systems Journal*, **13**(1), pp.115-139.

Stoustrup, B., (1986), *The C++ Programming Language*, Reading, MA: Addison-Wesley.

Sutcliffe, A., (1988), *Jackson System Development*, Englewood Cliffs, NJ: Prentice-Hall.

Swartout, W. & Balzer, R., (1982), "On the Inevitable Intertwining of Specification and Implementation", *Communications of the ACM*, July, pp.438-440.

Taenzer, D., Ganti, M. & Podar, S., (1989), "Problems in Object-Oriented Software Reuse", in *ECOOP'89 Proceedings of the Third European Conference on Object-oriented Programming*, Nottingham, UK, 10-14 July, (ed. S. Cook), Cambridge University Press, pp. 25-38.

Taenzer, D., Ganti, M. & Podar, S., (1989), "Problems in Object-oriented Software Reuse", in *ECOOP'89 Proceedings of the Third European Conference on Object-oriented Programming*, Nottingham, UK, 10-14 July, (ed. S. Cook), Cambridge University Press, pp.25-38.

Tello, E. R., (1989), *Object-oriented Programming for Artificial Intelligence: A Guide to Tools and System Design*, Reading, MA: Addison-Wesley.

Tesler, L., (1985), "Object Pascal Report", *Structured Languages World*, **9**(3), pp.10-14.

Thatte, S. M., (1986), "Persistent Memory: A Storage Architecture for Object-oriented Database Systems", in *Proceedings of the 1986 International Workshop on Object-oriented Database Systems*, Washington, DC, USA, IEEE Computer Society Press, **24**(10), pp.148-159.

Thomas, D., (1989), "In Search of an Object-oriented Development Process", *Journal of Object-oriented Programming*, May/June, **2**(1), pp.61.

Tsichritzis, D. C. & Nierstrasz, O. M., (1988), "Fitting Round Objects into Square Databases", in *ECOOP'88 Proceedings of the Second European Conference on Object-oriented Programming*, Oslo, Norway, 15-17 Aug., (ed. S. Gjessing & K. Nygaard), Lecture Notes in Computer Science, No.322, Springer-Verlag, pp.283-299.

Tsiritzis, D. C. & Lochovsky, F. H., (1982), *Data Models*, Englewood Cliffs, NJ: Prentice-Hall.

Turner, D., (1986), "An Overview of Miranda", *ACM SIGPLAN Notices*, **12**(12), pp.158-166.

Turner, J. A., (1987), "Understanding the Elements of System Design", in *Critical Issues in Information Systems Research*, (eds. R. S. Boland, Jr. & R. A. Hirschheim), Chichester: Wiley, pp.97-111.

Vlissides, J. & Linton, M., (1988), "Applying Object-oriented Design to Structured Graphics", in *Programming of USENIX C++ Conference*, California: USENIX Association, pp.93.

Walker, I. J., (1992), "Requirements of an Object-oriented Design Method", *Software Engineering Journal*, March, pp.102-113.

Ward, P. & Mellor, S., (1986), *Structured Development of Real-Time Systems*, New York: Prentice-Hall.

Ward, P. T. & Mellor, S. J., (1986), *Structured Development for Real-Time Systems*, Yourdon Press.

Wasserman, A. I. & Pircher, P. A., (1991), "From Object-oriented Analysis to Design", *Journal of Object-oriented Programming*, Sept., pp.46-50.

Wasserman, A. I. & Pircher, P. A., (1991a), "The Spiral Model for Object Software Development", *Hotline on Object-oriented Technology*, **2**(3), Jan., pp.8-12.

Wasserman, A. I., Pircher, P. A. & Muller, R. J., (1989), "An Object-Oriented Structured Design Method for Code Generation", *ACM SIGSOFT* , **14**(1), pp. 32-55.

Wasserman, A. I., Pircher, P. A. & Muller, R. J., (1989), "An Object-orientedStructured Design Method for Code Generation", *ACM SIGSOFT Software Engineering Notes*, **14**(1), pp.32-55.

Wasserman, A. I., Pircher, P. A. & Muller, R. J., (1990), "The Object-orientedStructured Design Notation for Software Design Representation", *IEEE Computer*, March, pp.50-62.

Wasserman, A. I., Pircher, P. A. & Muller, R. J., (1990), "The Object-Oriented Structured Design Notation for Software Design Representation", *IEEE Computer*, March, pp. 50-63.

Wegner, P., (1987), "Dimensions of Object-Based Language Design", in *OOPSLA'87 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Orlando, Florida, USA, 4-8 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., 22(12), pp.168-181.

Wegner, P., (1987), "Dimensions of Object-based Language Design", in *OOPSLA'87 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Orlando, Florida, USA, 4-8 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **22**(12), pp.168-182.

Wiener, R. & Sincovec, R., (1984), *Software Engineering with Modula-2 and Ada*, Chichester: Wiley.

Wilson, D. A., (1989), "Class Diagrams: A Tool for Design, Documentation, and Teaching", *Journal of Object-oriented Programming*, Jan./Feb., pp.38-44.

Wilson, D., A., (1990), "Class Diagrams: A Tool For Design, Documentation, and Teaching", *Journal of Object-Oriented Programming*, Jan./Feb., pp.38-44.

Winblad, A. L. & Edwards, D. S., (1990), *Object-oriented Software*, Reading, MA: Addison-Wesley.

Winblad, A. L., Edwards, S. D. & King, D. R., (1990), *Object-oriented Software*, Reading, MA: Addison-Wesley.

Wing, J. M. & Nixon, M. R., (1989), "Extending Ina Jo with Temporal Logic", *IEEE Transactions on Software Engineering* , Feb.

Winston, P. H. & Horn, B. K. P., (1981), *LISP*, Reading, MA: Addison-Wesley.

Wirfs-Brock, R. & Wilkerson, B., (1989), "Object-oriented Design: A Responsibility-Driven Approach", in *OOPSLA'89 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, New Orleans, Louisiana, USA, 1-6 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, **24**(10), pp.71-76.

Wirfs-Brock, R. J. & Johnson, R. E., (1990), "Surveying Current Research in Object-oriented Design", *Communications of the ACM*, Sept., Sept., **33**(9), pp.104-124.

Wirfs-Brock, R., Wilkerson, B. & Wiener, L., (1990), *Designing Object-oriented Software*, Englewood Cliffs, NJ: Prentice-Hall.

Wirth, N., (1986), *Algorithms and Data Structures*, Englewood Cliffs, NJ: Prentice-Hall.

Wolczko, M. I., (1988), *Semantics of Object-oriented Languages*, PhD Thesis, The University of Manchester.

Wolczko, M., (1992), "Encapsulation, Delegation and Inheritance in Object-oriented Languages", *IEEE Software Engineering Journal*, March, **7**(2), pp.95-101.

Yeh, R. T., (1990), "An Alternative Paradigm for Software Evolution", in *Modern Software Engineering: Foundations and Current Perspectives* , (eds. P. A. Ng & R. T. Yeh), New York: Van Nostrand Reinhold, pp.7-22.

Yonezawa, A. & Tokoro, M. (eds.), (1987), *Object-oriented Concurrent Programming*, Massachusetts: MIT Press.

Yonezawa, A., Briot, J.-P. & Shibayama, E., (1986), "Object-oriented Concurrent Programming in ABCL/1", in *OOPSLA'86 Object-oriented Programming Systems, Languages and Applications Conference Proceedings*, Portland, OR, USA, 29 Sept.-2 Oct., (ed. N. Meyrowitz), ACM SIGPLAN Notices, Nov., **21**(11), pp.258-268.

Yourdon, E. & Constantine, L. L., (1979), *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Englewood Ciffs, NJ: Prentice-Hall.

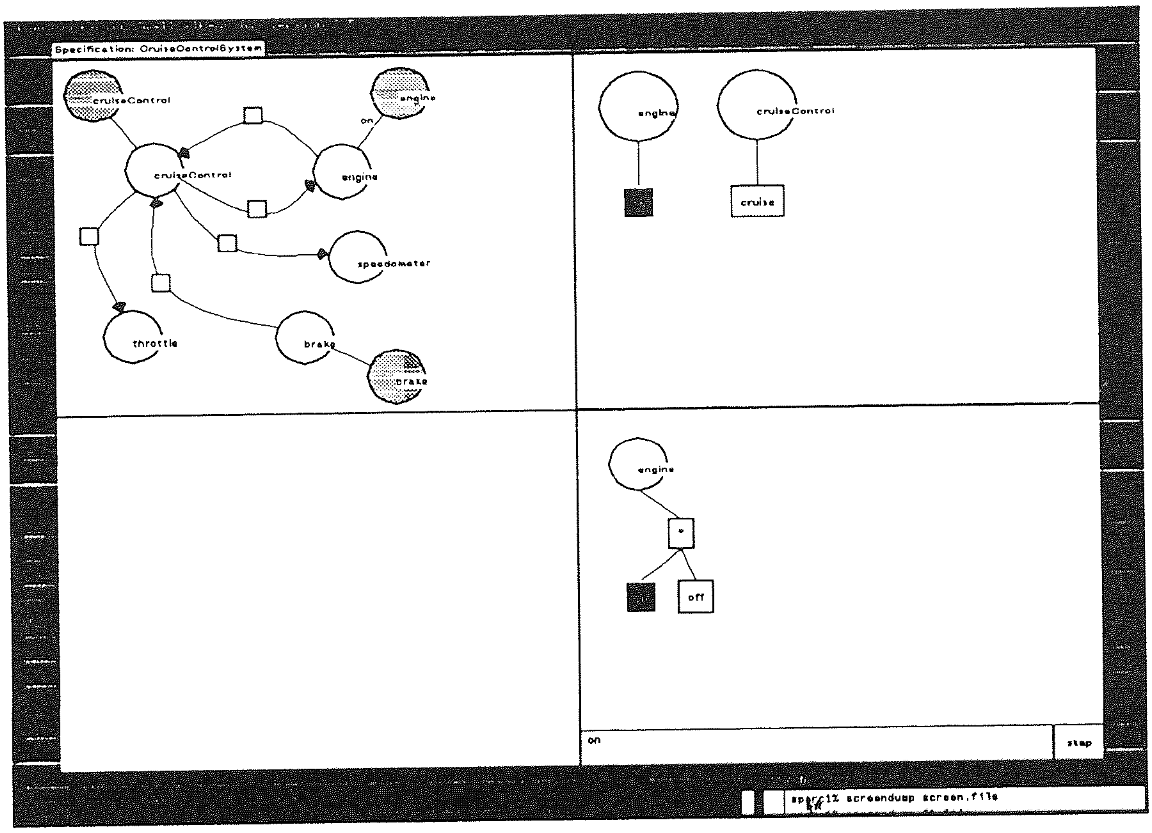
Yourdon, E. & Constantine, L., (1979), *Structured Design*, New York: Prentice-Hall.

Zave, P., (1982), "An Operational Approach to Requirements Specification for Embedded Systems", *IEEE Transactions on Software Engineering*, **SE-12**(2), pp.241-250.

Zave, P., (1984), "The Operational Versus the Conventional Approach to Software Development", *Communications of the ACM*, Feb., **27**(2), pp.104-118.

Zave, P., (1984), "The Operational Versus the Conventional Approach to Software Development", *Communications of the ACM*, February, pp.104-118.

Appendix A Snapshots showing execution animation of the automobile cruise control specification

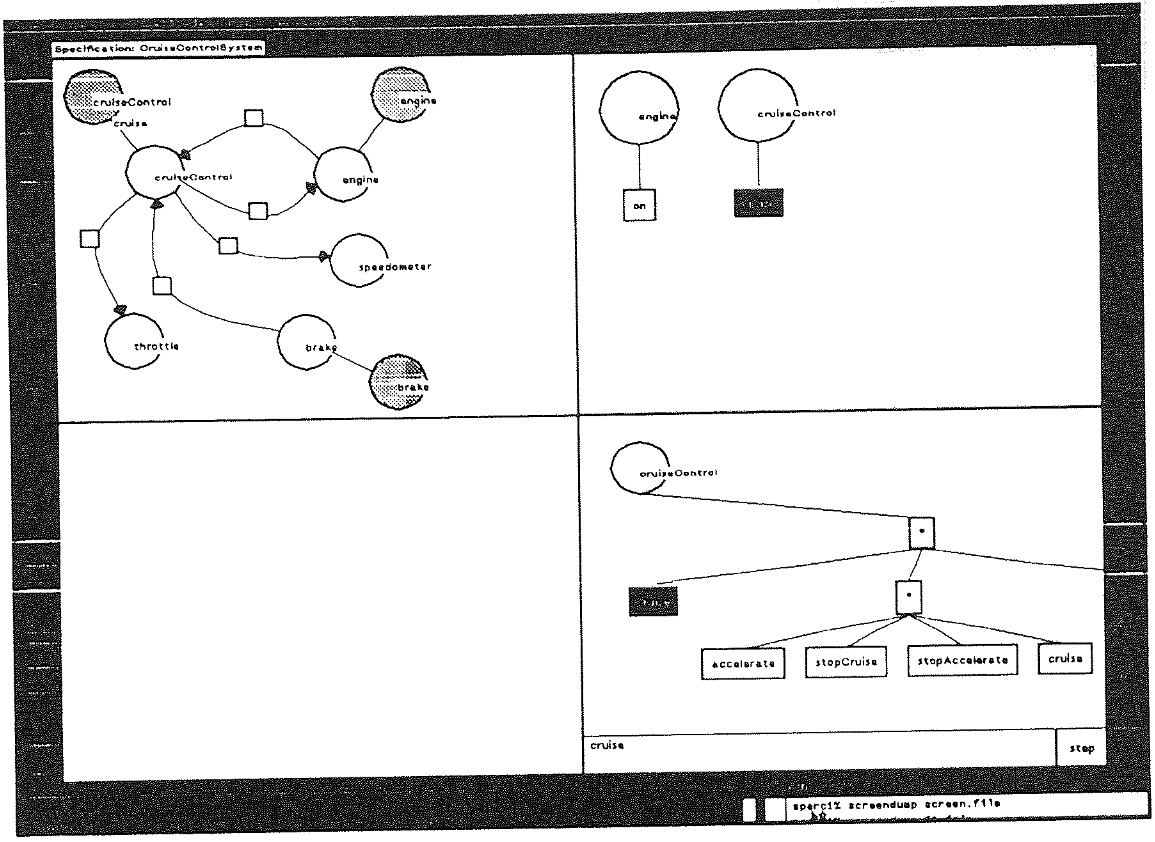


on

```
sparc1% screendump screen.file
```

Figure A.1 engine receives the message on.

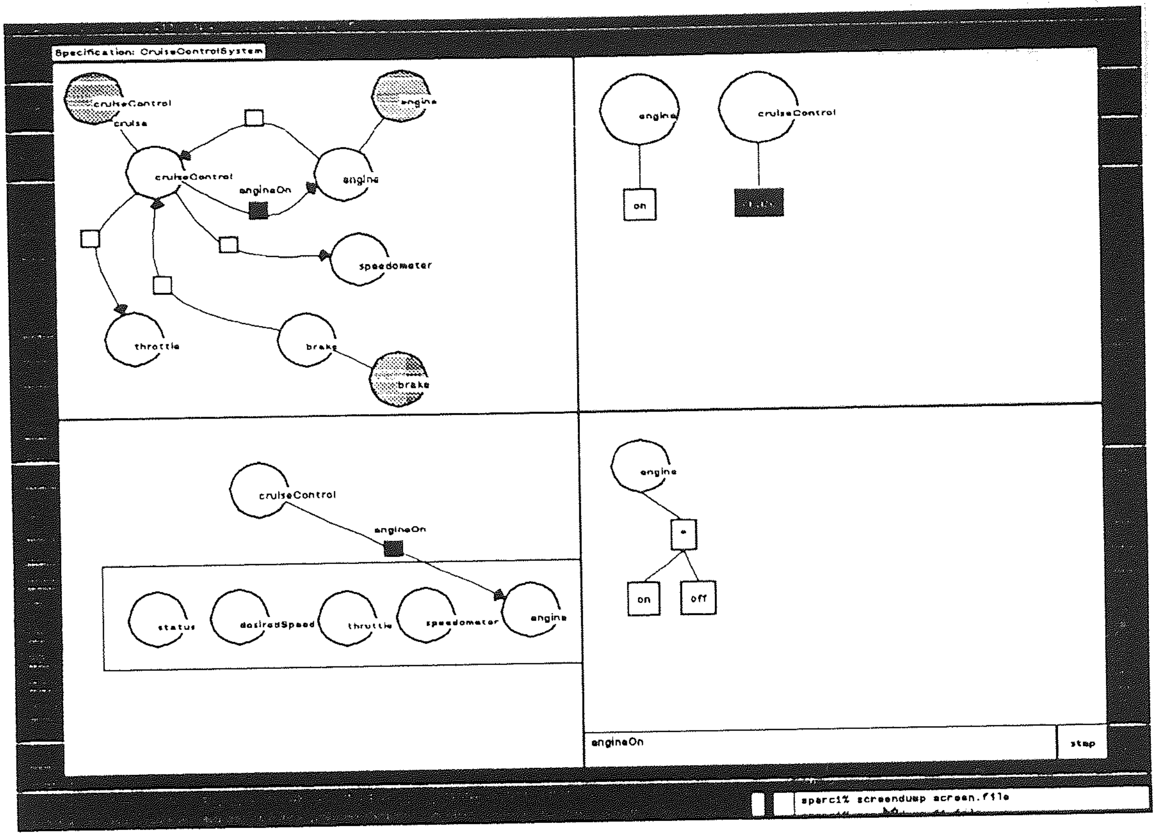
The engine object receives the message on from the user. on is the first entry in the iteration of the object's life history.



```
sparc1% screendump screen.file
```

Figure A.2 `cruiseControl` receives the `cruise` message.

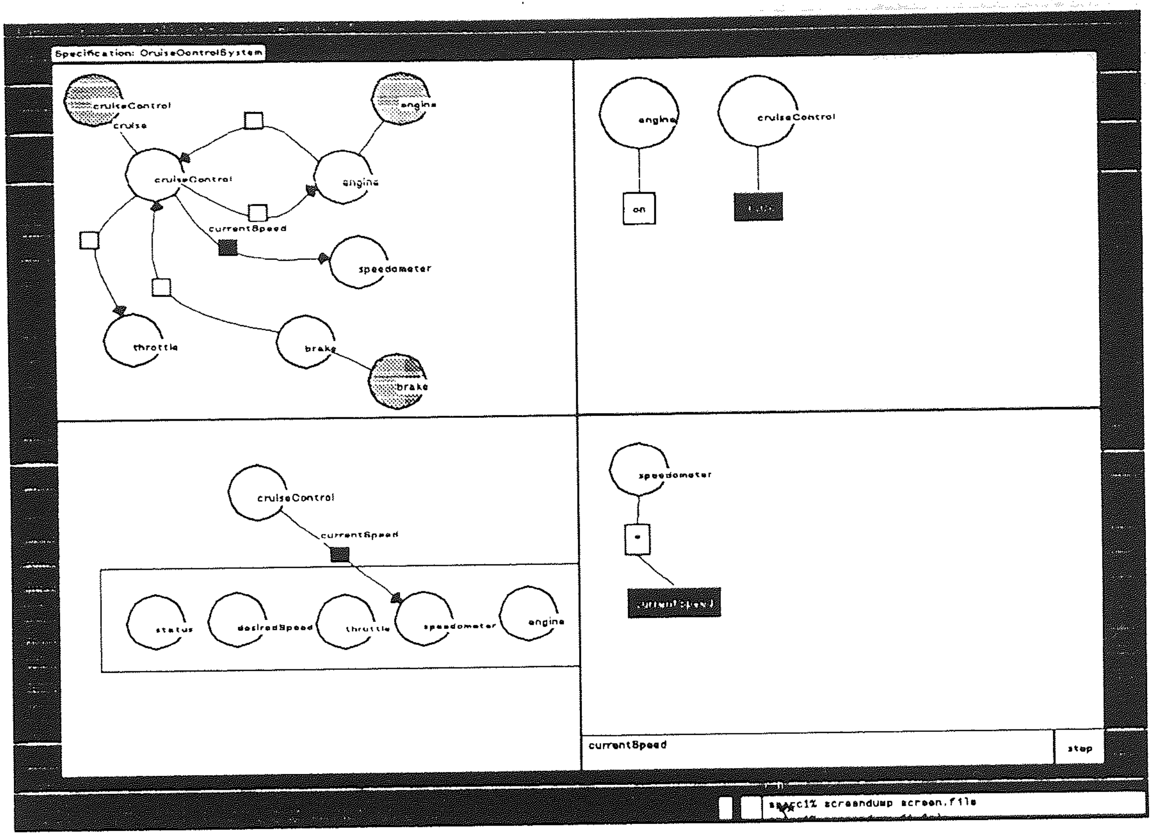
The `cruiseControl` object receives the `cruise` message from the user. `cruise` is the first entry in the overall iteration of the object's life history.



```
sparc1% screendump screen.file
```

Figure A.3 `cruiseControl` sends the message `engineOn` to `engine`.

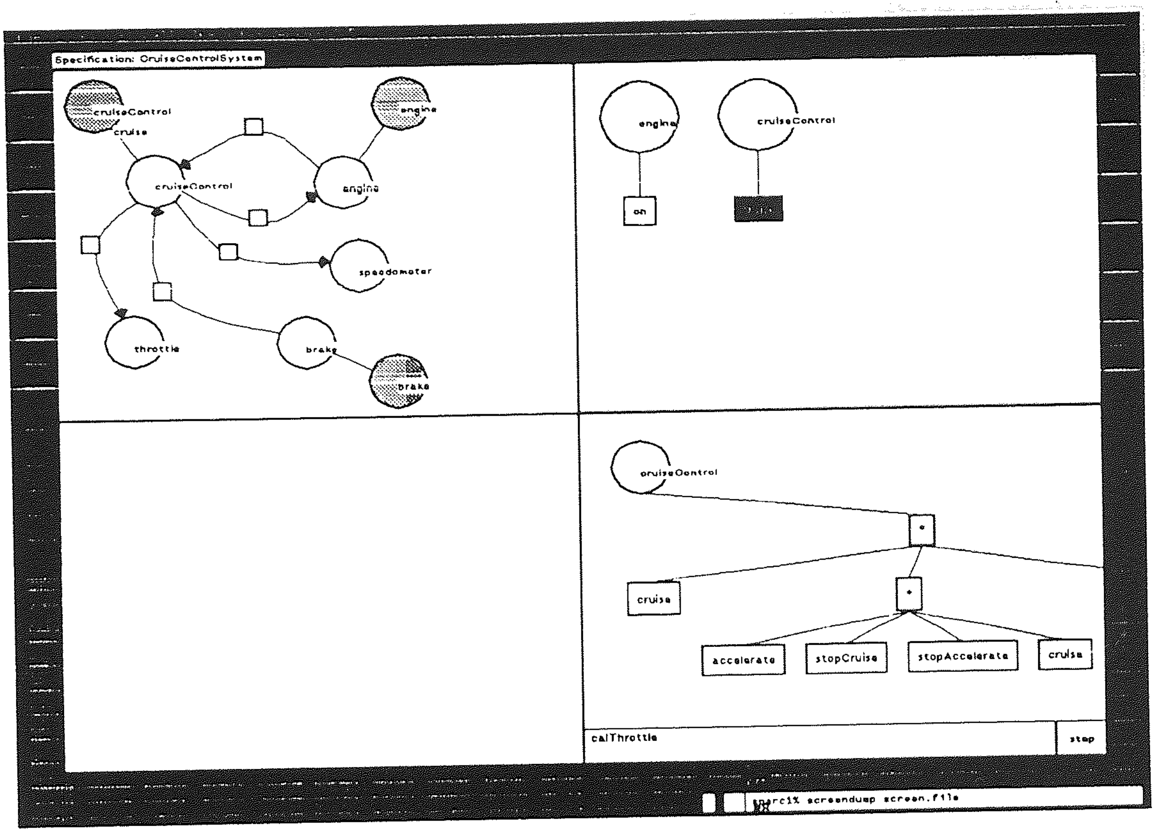
`cruiseControl` needs to check that the engine is on before starting its cruising function. The message `engineOn` does not have any constraint on its invocation and so does not appear in the `engine` object's life history.



```
sparc1% screendump screen.file
```

Figure A.4 `speedometer` receives the `currentSpeed` message from `cruiseControl`.

The current speed is obtained from the `speedometer` object and used as the cruising speed. The `currentSpeed` message is the only entry in the iteration of the `speedometer` object's life history.

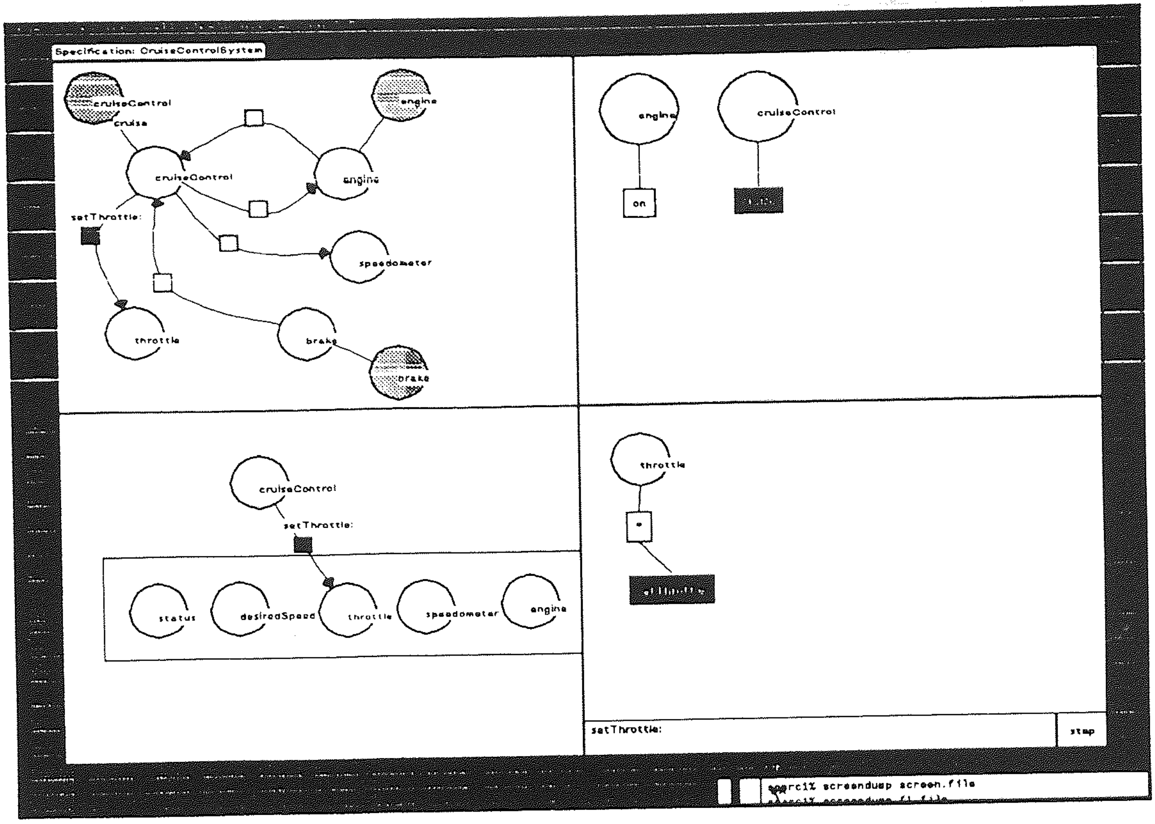


calThrottle

```
sparc1% screendump screen.file
```

Figure A.5 `cruiseControl` sends itself the message `calThrottle`.

`cruiseControl` sends itself the message `calThrottle` to calculate the required throttle position to maintain the cruising speed based on the current speed returned by `speedometer`. `calThrottle` is not contrained in `cruiseControl`'s life history.



```
sparc1% screendump screen.file
```

Figure A.6 Throttle receives setThrottle: from cruiseControl.

The last task to achieve cruising involves cruiseControl sending the required throttle setting to the throttle object. The message setThrottle: is the only entry in the iteration of the throttle object's life history.