Object-oriented design: guidelines and techniques

L F Capretz and P A Lee*

The paper is concerned with object-oriented design methodologies for software systems. A general methodology for object-oriented design, called MOOD, is presented. MOOD is unrelated to any programming language, yet is capable of being used to design a variety of object-oriented software systems. In particular, MOOD allows the creation of a design mainly in terms of classes, objects and inheritance, and the representation of a design graphically by a set of class hierarchy diagrams, composition diagrams, object diagrams and operation diagrams.

object-oriented design, object-oriented methodology, object-oriented diagram, software engineering

In the past few years, demand for object-oriented software systems has increased dramatically, and it is widely accepted that present software engineering methodologies are unable to cope with the needs of that demand. The object-oriented paradigm has promised to revolutionize software development, and it has been seen as an attempt to extend and apply the techniques of encapsulation and inheritance, not only in the implementation phase but also during the design and system analysis phases of the software development process. As a result, several methodologies have recently arisen to support software development based on an object-oriented approach.

Currently, the most widely used software engineering methodologies are those for structured development. Such methodologies are popular because they are applicable to many types of application domains. On account of this popularity, structured development methods have been mixed up with an object-oriented approach. Recently, there has been a profusion of object-oriented methodologies for analysis and design influenced by a variety of different backgrounds. Many approaches are evolving existing structured methods into object-oriented methodologies, but as a result bringing their limitations with them.

Designers who come from a traditional software engineering background, such as functional decomposition and data modelling techniques, will probably find the methodologies of Shlaer and Mellor[1], and Coad and Yourdon[2] familiar because these methodologies are clearly adaptations of traditional structured development methods and data modelling techniques. These methodologies may be used during a period of transition from structured development to an object-oriented approach as a compromise.

This tendency can also be clearly seen in the early version of Booch[3] methodology and its successors, such as Seidewitz[4], Heitz[5] and Jalote[6]. These methodologies do not make an adequate distinction between the definition of classes and use of objects, which is essential for exploitation of the object-oriented paradigm. Similarly, they have offered limited support for inheritance of commonalities in a hierarchy of classes; they tend to be oriented to Ada notations of *package* and *task*, rather than to more general notions in object-oriented design.

Other methodologies are trying to put the object-oriented paradigm within the traditional waterfall software life-cycle model. In recent years several object-oriented methodologies have appeared but they only partially cover a software life-cycle. Several authors have tried to fit the object-oriented paradigm into this framework: Booch[7], Wirfs-Brock[8], Wasserman[9] and Rumbaugh[10] can be considered as good examples. These methodologies are well known and generally accepted within the object-oriented community; their main ideas encompass object-oriented concepts because they are based on at least classes, objects and inheritance. Nevertheless, these methodologies are still at a relatively early stage of growth. It is clear that even more experimentation is required, particularly in developing substantial software systems, before they can claim to be mature methodologies.

Therefore, it can be said that so far there has been no widely accepted object-oriented design methodology. Moreover, a revolution is what is needed to tackle the problems of software development. New methodologies which exploit the benefits of the object-oriented paradigm within a substitute software life-cycle model have to be pursued. As a result, it is the intention of this paper to present an object-oriented design methodology which allows designers to apply powerful object-oriented principles to the design of a wide range of applications from the beginning of software development. This paper is interested in an approach which yields a single coherent object-oriented design methodology, rather than separate methods to solve specific parts of a design. Such a

Department of Computer Software, University of Aizu, Aizu-Wakamatsu, Fukushima, 965 Japan
*Computing Laboratory, University of Newcastle upon Tyne, Newcastle upon Tyne, NE1 7RU, UK

methodology must pay attention to object-oriented concepts, for instance, classes, objects and inheritance. The proper use of these concepts can lead to a truly general object-oriented design methodology as independent as possible from any programming language.

Because of the rapid developments in the object-oriented field, it has become very fashionable to describe many kinds of software system using object-oriented terminology, and the term itself has become a buzzword. In the scope of this paper, an **object** embodies an abstraction characterized by an entity in the real world. A **class** (or type) is a template description which specifies common properties and behaviour for a group of similar objects and an object is an instance of a class. The classes themselves can be organized into class hierarchies. Such class hierarchies allow similar classes to be related together in such a way that commonalities of one class can be inherited (reused) rather than duplicated by classes lower in the hierarchy, thus simplifying the design and implementation of these lower level classes.

The properties and behaviour of objects, and hence their commonalities, are described in terms of **attributes** and **operations**. An attribute is a named property of an object which holds a value and maintains an abstract state for that object. An operation identifies an action which may be applied to objects of a class. Objects from each class are manipulated by invoking the operations upon the attributes of these objects. **Inheritance** is a mechanism which permits classes to share attributes and operations based on relationships of specialization and generalization between them within a hierarchy of classes.

It has been claimed in the literature that software systems developed using an object-oriented approach can be significantly more elegant than those developed using traditional structured development approaches, and that more software components can be reused during software development. However, using an object-oriented programming language does not, by itself, guarantee miraculous results. Like any other engineering activity, methodologies play an important role during the design of object-oriented software systems.

This paper is aimed at the creation of a Methodology for Object-Oriented Design (MOOD), which, in contrast to other methodologies, takes reusability into account as an important aspect of the software life-cycle. The main characteristic obtained through the use of MOOD is the design of a software system following strictly object-oriented concepts, as MOOD concentrates on identifying and representing classes, inheritance and objects. In doing so, the architecture of an object-oriented software system at the design level is built around sets of classes and objects. MOOD supports the design of a software system following an object-oriented approach and it is independent of the idiosyncrasies of any particular programming language. Additionally, considering reusability as a pragmatic (and desirable!) process within the design phase helps the designer to relate software components to each other through relationships which show where a component is defined and used, and in what context. In this way, reusability is considered within a software life-cycle model as part of the proposed object-oriented design methodology.

MOOD is based on graphical representations of classes, objects and inheritance, with associated rules, principles and guidelines which facilitate the identification and representation of software systems in terms of these classes, objects and inheritance. MOOD employs steps which help the designer to identify classes, build class hierarchies using inheritance, describe the software system behaviour with objects and represent a design graphically with several kinds of diagrams. The rigorousness of the proposed notation increases as the steps are carried out because the design process starts from a given abstract model of the application, often informal, and ends with a detailed object-oriented graphical representation of the software system.

MOOD is aimed at designing software systems by following prescribed steps which allow the designer to represent and describe software systems at different levels of abstractions. A design using MOOD basically starts from a given system analysis and produces a graphical description to be implemented, which comprises an *information model* and a *behaviour model*, together called the **design model**. The information model is a static representation of a software system using a set of diagrams which shows a global view of the classes and the class hierarchies, built during a stage named *static design*. The behaviour model shows the dynamic relationships between objects, created during a stage denominated *dynamic design*. The design model enables MOOD to maintain close links between the system analysis, design and implementation phases of the software life-cycle, so that the gap which often exists between these three phases can be bridged. In this way, MOOD supports traceability from system analysis through to implementation, and this is believed to be another important aspect of MOOD.

Whereas the basic object-oriented steps and concepts in MOOD appear similar to features in other object-oriented design methodologies, it is MOOD's emphasis on reusability and software life-cycle issues which differentiate it from others, as it is discussed by Capretz and Lee[11]. However, many of the general issues, while described in the context of MOOD, are relevant to the application of other object-oriented design methodologies.

The remainder of this paper is divided into three sections. The second section places MOOD into the context of software development. The steps which must be followed in order to design a software system using MOOD are discussed in the third section. This section concludes with final remarks on MOOD, outlining some of the issues which should be considered when designing and representing an object-oriented software system. Finally, the last section presents some conclusions related to experiences with the use of MOOD.

THE CONTEXT OF MOOD

The designer of a software system should start the design from an abstract model of the application, which is established through system analysis. This abstract model is the first representation of the software system. But what should the system analysis be? System analysis is characterized by obtaining information about the application, and hence this information is non-structured, often incomplete and sometimes contradictory. The system analysis then produces a description in terms of the requirements and objectives of the software system which can be refined through the addition of details to that abstract model.

Methodologies which aim to give support for system analysis should consider the possibility of dealing with an incomplete abstract model and describing partial aspects of the application, then refining and complementing that abstract model. The result of system analysis comes as a graphical or textual, informal or formal, abstract model of the application. The more complete and consistent it is, the better. Therefore, system analysis is a means for understanding the application. MOOD commences with an abstract model of the application as input and supports the production of a design model as output. The design model comprises an information model and a behaviour model, as introduced earlier. Figure 1 shows, within the dashed rectangle, the context of MOOD. Although the boundary between each representation is usually fuzzy, the abstract model of the application, the design model and the program are three different, yet related abstractions of the same software system. The object-oriented paradigm allows the designer to create abstractions that are close to the application, and to manipulate these abstractions throughout software development. Therefore, within an object-oriented framework, it is even more difficult to draw a distinct line between system analysis, design and implementation.

As far as the reusable library is concerned (see Figure 1), this contains a collection of reusable components, from both application and solution domains, put into and taken from there during software development. The use of a reusable library aims to identify components which can be reused in the developing software system. A reusable library requires a good way of classifying, storing and recovering components. The reuse of such components can take place in two ways: either directly when the component corresponds exactly to that required, or through inheritance when there are some differences, and specialization and/or generalization are necessary.

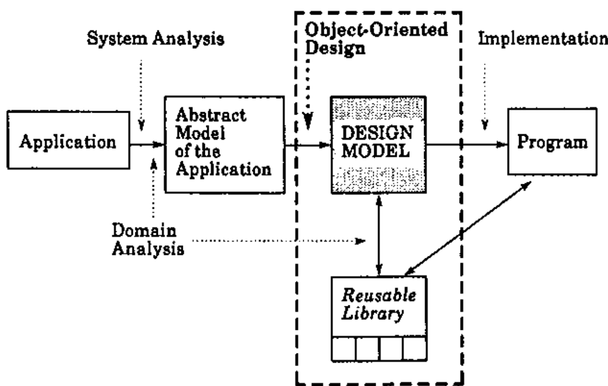

Figure 1. The context of MOOD

Most software systems are concerned with an application domain, and the identification of the vocabulary for the application domain is the main role of domain analysis. Domain analysis also plays a fundamental role in identifying potentially reusable components within an application domain because it helps the designer to establish a vocabulary for a given application domain and therefore components which are related to software systems within that application domain. A comprehensive hierarchy of classes for an application domain provides the designer not only with application domain reusable components, but also potential parts for a software system. The essence of a good object-oriented design is design with reuse as well as design for reuse. That means finding application domain reusable components which can be taken from reusable libraries as well as producing solution domain reusable components which may be stored in a library.

The designer cannot be expected to have a perfect understanding of the software system at the beginning of the design. Rather, that understanding evolves through iterations and refinements with constant feedback, but each iteration makes the design model increasingly clear. A top-down fashion for software development generally creates the software system by successive refinements of software components. Nevertheless, it is not easy for the designer to divide a software system into components if the software system is almost unknown. It could be argued that the top-down decomposition only works when the designer already knows the application domain. Otherwise, a bottom-up approach would be more appropriate. These issues, along with an alternative object-oriented software development life-cycle model and an environment which supports MOOD, are further discussed in Capretz[12], but for now the MOOD steps are introduced.

THE STEPS FOR OBJECT-ORIENTED DESIGN

In order to produce the design model following an object-oriented approach, it is necessary to identify and represent:

- software system classes
- inheritance between classes
- software system behaviour in terms of object interactions

An important aspect of MOOD is how the designer tackles the problem of designing in successive steps in order to produce the design model to be implemented

subsequently. Briefly, the fundamental steps proposed by MOOD (and other methodologies) are:

- divide the software system into manageable components
- identify classes and/or objects which model the application
- identify inheritance between classes
- represent classes and inheritance
- identify software system behaviour in terms of objects and operations
- represent software system behaviour in terms of object interactions

The MOOD approach to tackling these steps is discussed in the following subsections. It is important to note that the sequence in which these steps are carried out depends on the knowledge that the designer has about the application domain.

Representation of the design model

Graphical notations have been an integral part of most software engineering techniques. In fact, there are very few software engineering activities which do not benefit from some form of graphical notation. Their use has proved to be an effective mechanism for expressing a design because a clean graphical notation can show the architecture of a software system more clearly than a textually-based notation.

A systematic approach to software design can be offered by a set of guidelines supported by a graphical notation used to represent a software system. A good graphical notation should be simple, straightforward and a reflection of the paradigm used to build the software system. Thus, a graphical notation chosen for object-oriented design should directly support the object-oriented paradigm, by providing representation for classes, objects and inheritance.

An important aspect of MOOD is the graphical notation which allows a straightforward visual representation of an object-oriented design. The graphical notation to be presented later in this paper comprises different types of diagrams which depict the features of a generic object-oriented representation that goes hand in hand with MOOD. The diagrams are independent of any programming language, provide a high level representation of a software system design and span a broad range of object-oriented concepts without making any assumptions about implementation. The guiding principle behind the proposal for these diagrams has been to keep the diagrams as simple as possible. There are four main types of diagrams:

- **Composition diagrams:** these represent the composition and decomposition of a software system in terms of its components.
- **Class hierarchy diagrams:** these are a simple but effective way to display classes, their attributes and operations, and inheritance relationships.
- **Object diagrams:** these show relationships among objects based on requests for operations between them.
- **Operation diagrams:** these depict how operations are combined to provide particular software system functionality.

These diagrams are the principal means for representing the design using MOOD. They constitute the graphical representation of the software system design as a whole and can be viewed as communicating meaning between designers. The diagrams are composed of simple symbols, and their automated support in a computer-aided software engineering (CASE) environment is presented in Capretz[12]. The number of different basic symbols is small, the symbols are unambiguous and the visual impact of the arrangements of the basic symbols connotes the semantics of object-oriented concepts.

These diagrams are integrated with each other, and the information present in a particular diagram must be consistent with the information used in another diagram. For instance, the operations used in a certain operation diagram must be defined in any class of another class hierarchy diagram. Such inter-relationships between diagrams, which are automatically checked by MOOD, makes them a powerful collection of graphical notations capable of representing a whole design. The following subsections show how to build up these types of diagrams which are the means by which the designer represents the software system at design level when MOOD is used.

Identification of components

The development of large software systems imposes some characteristics on the design process. Often, the design of a software system of any significant size must be divided among different designers or several designers grouped into small teams. A large software system will probably require numerous components which make such a software system difficult to comprehend.

When should identification of components be introduced? This depends on the services provided by the software system. Large components are more likely to be identified during the early stages of a design and can be further divided into subcomponents. In a small software system, where just a few services are provided, identification of components may not be necessary. However, a large and/or complex software system, providing many different and complex services, needs decomposition to be applied from the beginning of the design. Nevertheless, finding a good division into components for an object-oriented software system still seems to be a subjective process, dependent upon the experience (and whims) of the designer.

Composition diagrams

Composition diagrams can be used to represent all or part of a software system in terms of its components.

Composition diagrams show aggregations of components, where components can be any software item such as subsystems, modules, classes or objects which make up the software system. A component may also be subdivided into other, smaller subcomponents, and vice versa; for instance, interdependent and co-operating components can be composed to offer a particular service.

Each component in a composition diagram fulfils the role of a software item in terms of the services provided by that component. Figure 2 illustrates a composition diagram in which a CASE environment software system can be composed of three different components representing the tools (*graphiceditor*, *texteditor* and *checkers*) of the environment. That figure also shows how the *checkers* can be composed of its subcomponents (*consistency* and *completeness*). Each component is represented in the diagram by an ellipse. It is not the intention of composition diagrams to capture all the details of a software system; rather the purpose is simply to represent broadly the components of a software system. The relationship between large components and classes will be discussed later in this section.

Identification of classes

Classes are the most important concept for object-oriented design. Identification of classes in MOOD involves the recognition of important classes in the abstract model of the application. Nevertheless, there is no easy and fast way of defining what is and what is not a good class. Part of the identification process is to assess the consequences of including or excluding potential classes. An abstraction in terms of classes depends on the purpose the classes play in the software system as a whole.

Finding classes requires several iterations before a suitable collection of classes can be determined. Iterations are not a sign of bad design and should be regarded as a healthy process by which learning takes place. The number of iterations depends on the knowledge that the designer has about the application domain,

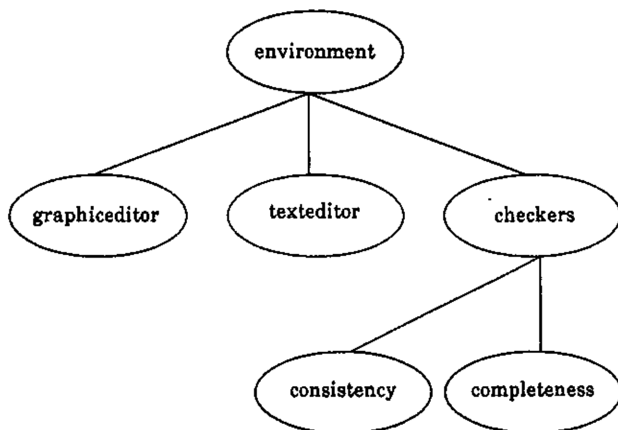

Figure 2. A composition diagram for an environment

and the intuition and skills of the designer, which are gained through insight and experience. As a result, it is only possible to propose guidelines which assist the designer in determining the classes for a particular design.

First, the designer needs to understand the concept of a class and what may be a class in the software system. Candidate classes can be:

- abstractions of things: e.g. books, planes, sensors
- abstractions of people: e.g. professors, students, engineers
- abstractions of concepts: e.g. departments, graphics, flights, accidents
- roles played by things, people or concepts: e.g. people are *voters* for politicians or *taxpayers* for the government
- relationships between abstractions of things, people and concepts: e.g. people *purchase* things, people *marry* people

As described in the earlier part of the paper, other important concepts concerned with a class are attributes and operations. As far as this step of MOOD is concerned, the designer should also consider the abstract model of the application and take into account services provided by a class, which can be identified as its operations. In order to identify the services, for each class, the designer should answer these questions:

- Which operations can be performed on instances of this class?
- Which actions do the attributes of this class undergo?

At the end of this stage, a set of classes (which provide abstractions for conceptual entities of the real-world application) and their attributes and operations should have been identified. Thus, the designer can view the software system design represented as a collection of classes. The graphical notation which has been developed to represent classes is discussed in the next subsection.

Identification of inheritance

Identifying classes is only the first step in designing an object-oriented software system. After some classes have been identified, they can be related to one another to form class hierarchies. The notion of inheritance plays a key role during object-oriented design because it helps the designer to derive new classes from primitive ones by exploiting their commonalities in terms of attributes and operations, and build up class hierarchies.

Inheritance enables the designer to create a new class simply by specifying the differences between the new class and an existing class instead of starting from scratch each time. As a design technique, the use of inheritance is similar to a *stepwise* refinement approach where inheritance divides the classes which model an application into hierarchies of classes. Thus, inheritance

can be used by defining a new class to be a specialization or generalization of existing ones.

Although formal or automated techniques for refining class hierarchies do not exist, there are a few rules which might help reorganize a collection of classes into hierarchies. The main rule for building a class hierarchy is to identify common attributes and operations and migrate them to a superclass, then eliminate from the superclass those operations that are frequently overridden in its subclasses rather than inherited by these subclasses. This rule makes the superclass more abstract and hence more generally useful. Over-riding and renaming of attributes and operations are permitted to adjust the subclasses to a particular context.

In order to identify inheritance, the designer must settle upon a collection of primitive classes from which all others can be derived by using the generalization and specialization mechanisms. A common example of inheritance by generalization is a *basic-screen-window* class which has been created as a superclass from the *coloured-window* and *framed-window* subclasses. Such a superclass embodies the minimal characteristic of all screen windows. The two subclasses could add greater functionality to the *basic-screen-window* class by defining such attributes as *foreground* and *background* colour, and *border-width*.

As far as inheritance by specialization is concerned, the designer should try to identify classes that do not properly describe (with enough details) all their instances. In this case two or more specialized subclasses can be derived. It is possible to specialize in the subclass the general properties defined in the superclass. A good example of inheritance by specialization is the following: consider a *vehicle* class, with attributes *licence-number*, *maker* and *model*. This class can be further specialised into *freighter*, *bus* and *car* subclasses. The *freighter* class could be specialized into *van* and *lorry* subclasses. The attributes of the *vehicle* class are inherited by its subclasses so that, for instance, a *van* object has the *licence-number* attribute (from the *vehicle* class), as well as a *permitted-load* attribute (from the *freighter* class) besides its private attributes. A *van* object could now be handled either as a *vehicle* object, as a *freighter* object, or as a *van* object; the different viewpoints imply restricted visibility of the attributes, and make it possible to handle all subclasses simply as a *vehicle* class, whenever that is desirable. Attributes and operations of the superclass are available to their subclasses and new attributes and operations can be defined within the subclasses.

After having identified the primitives classes which may form the basis for the software system, the designer can also use *set theory* in such a way that intersections between sets of attributes and operations of different classes can be singled out and inheritance identified. *Venn diagrams* are a convenient means of representing *sets* and can be used as a technique to help in the identification of inheritance as well. Such diagrams increase the understanding of inheritance by treating classes as sets of attributes and operations. Venn diagrams can give a hint of possible superclasses by showing which attributes and operations are held in common among classes (see Figure 3). If two or more classes have some attributes or operations in common, they could inherit them from a common superclass. If that common superclass does not already exist, the designer should create one and move the common attributes and operations to it.

The following kinds of changes to the collection of classes are to be expected during the evolution

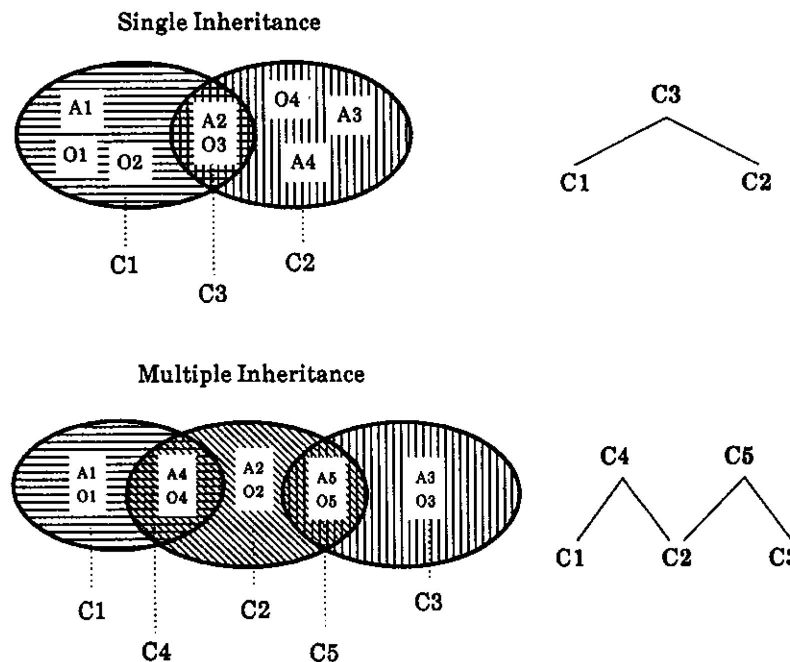

Figure 3. Identification of inheritance from Venn diagrams (A: attributes, O: operations, C: classes)

of a design, while the class hierarchies are being built:

- add new classes to the collection
- change the attributes and operations of a class
- reorganize the class hierarchy using the specialization and generalization mechanisms

The designer may change a class either to add or delete some attributes or operations. The designer can also define new classes when new key abstractions are discovered. The reorganization of the class hierarchies takes the form of changing inheritance relationships, adding new generic classes and shifting attributes and operations in the class hierarchies. Reorganization happens frequently at the beginning of the design and then (hopefully) stabilizes over time as the designer better understands the key abstractions.

Class hierarchy diagrams

Class hierarchy diagrams in MOOD show the existence of classes and their relationships in a design of a software system. Such diagrams depict how classes are arranged hierarchically. The designer can use class hierarchy diagrams to represent static aspects of the software system by using only the fixed set of symbols as illustrated in Figure 4. A rectangle represents a class and is annotated with its mnemonic name. A class name is indicated in a class hierarchy diagram followed by its attributes and operations. A class may restrict its attributes and operations from other classes by specifying them as *public* or *protected*. Any other attributes and operations in a class will be regarded as private, that is, part of the implementation details of that class and not visible in the class hierarchy diagrams.

A rectangle representing a class displays the externally visible (*public*) attributes and operations that instances of that class will possess. Subclasses derived from a superclass can, through the application of inheritance, make direct use of the public attributes and operations of the superclass. As well as public attributes and operations, the designer may wish to export additional attributes and operations which can only be used by subclasses. These attributes and operations of a class may be labelled *protected*, and will not be part of the normal interface of an object of that class. In the class hierarchy diagrams, protected attributes and operations are preceded by an asterisk ('*').

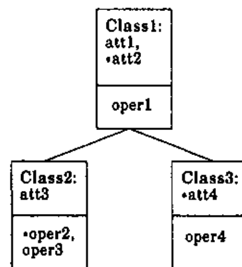

Figure 4. A generic class hierarchy diagram

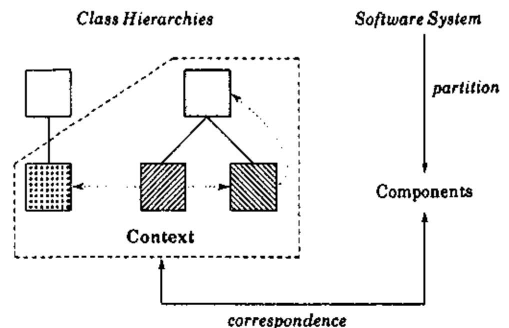

Figure 5. Correspondence between components and classes

Relationships between class hierarchies and components

It is possible to divide a software system following an object-oriented approach by partitioning it into independent components and making a correspondence between the services provided by each component and the operations of different classes in the class hierarchies. Large components identify functionality at a high level of abstraction, to be fulfilled by operations of different classes, and a class is associated with a component only if it contributes to the functionality provided by that component.

Large components will represent a collection of logically related classes, each providing different services which, when put together, provide the functionality required from the software system. A particular component has nothing to do with a specific class hierarchy, but with the functionality of the software system. In fact, composition diagrams are orthogonal to class hierarchy diagrams. Large components may use the services of several classes placed in different class hierarchies to contribute to the overall functionality of the software system.

The correspondence between large components and classes defines a **context** where functionality is provided by a set of operations of different classes, as depicted, within the dashed lines, in Figure 5 (in this figure, the dotted lines represent links between operations on objects of different classes). A context consists typically of a set of classes, connected together by inheritance relationships and a well-defined pattern of interaction between the objects of these classes. A context groups classes together so that a portion of the overall system functionality can be provided. Contexts will maintain traceability between classes and functionality; indirectly, a context shows the classes related to a component.

Identification of objects

This step is concerned with objects by focusing on when and how they are created, destroyed, accessed and changed. Some characteristics of object-oriented design regarding objects are:

- objects are instances of some class
- objects can be created and destroyed

- operations are related to objects
- attributes are encapsulated into objects so that other objects can only access them via operations
- requests are sent between objects to invoke operations or return results
- functionality can be provided by operations on several objects

Concrete entities in an application domain, such as people and things, are very likely to be objects in the software system. Therefore, they should be the initial targets to be identified as objects. Identification of objects involves:

- understanding the application behaviour
- depicting its main entities as objects
- identifying the attributes of an object
- identifying the operations on an object
- focusing on interaction between the objects

Identification of objects is also important because it helps define the dynamic behaviour of the software system in terms of object instantiation. The control flow of the operations, which defines the sequence in which the operations are requested, can also be identified. The representation of software system behaviour consists mainly of a network of objects manipulated by operations. Each object has its own internal states and is linked to the network through requests that establish the sequence of the operations.

In order to understand the interaction among objects, it is necessary to:

- identify the operations requested by an object
- identify the operations provided by an object
- identify the relationships between the objects
- identify the operations that can change the state of an object

The states that an object can go through can be represented in a *state transition diagram.* Such a diagram shows the states of an object and the events which cause a transition from one state to another. A state transition diagram can depict the behaviour of an object over a period of time through the states which an object assumes and the events which cause that change of state. The designer should look at the objects, viewing them as state machines and set the events applicable to each state. State transition diagrams may be used as a technique which can help the designer to identify the operations upon objects because the events that provoke the change of state of an object can be related to operations on that object. Each event may have conditions associated with it, and these conditions must be satisfied in order to have an operation triggered.

Figure 6 represents a state transition diagram for a *screen-window* object. In this figure, the labels in bold represent the various states which a *screen-window* object can go through. The labelled arrows represent events and possible operations on that *screen-window* object, and the effect that these events have on the state of that object. Some operations cause a change of state only when the object is in a certain state, for example the *edit* operation has no effect when a *screen-window* object is closed. These constraints help the designer to understand the behaviour of the software system.

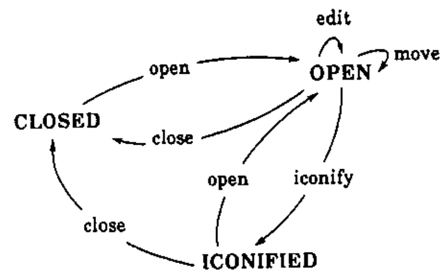

Figure 6. A state transition diagram for a screen-window object

Object diagrams

The graphical notation to be introduced in this subsection helps the designer to capture the behaviour of a software system by using object diagrams. Such diagrams show objects and the requests for operations on other objects. A request relates one object to another when one object requires another object to perform an operation. A request also depicts the dependency between a client object (the one which requests an operation) and a server object (the one which performs that operation). An object diagram is simply a network in which the nodes represent objects, and arcs connecting them represent operations; the interactions between objects happen one at a time.

Figure 7 shows an example of an object diagram. An object is represented by a circle containing its identification and the class name from which the object has been instantiated. Each object has a unique identifier that allows it to be referenced unambiguously within its class. A line entering an object represents a provided operation and a line leaving an object represents a requested operation. Object diagrams partially define the overall pattern of communication in the software system because they show which objects request which operations from other objects.

The object diagram represented in Figure 7 conveys the behaviour of some example objects when a hypothetical tool is used in a CASE environment. When an *atool* object manipulates an *awindow* object, the *awindow* object in turn can display a warning message *awarn* object and gets its state updated in a *a*window manager *amanager* object. When an *atool* object is used, it can open an *awindow* object, which can also be moved, edited, closed and iconified. An *awindow* object can also set a warning message *awarn* object to appear on the screen and this disappears after an *ok* operation occurs. A window manager *amanager* object can receive a request to update the position of an *awindow* object. It can also be seen that a client object

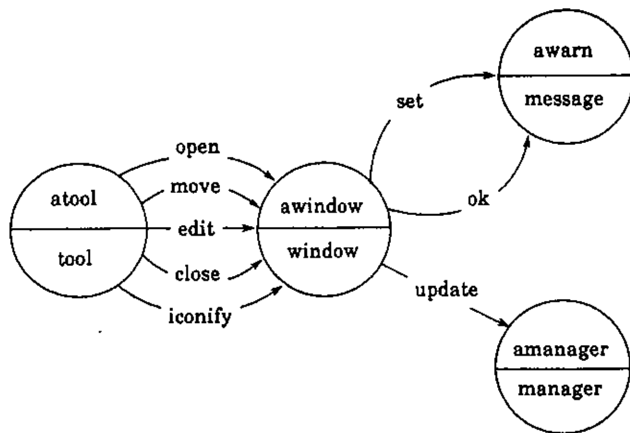

Figure 7. An object diagram for objects in an environment

(an *atool* object) controls the requests to another server object (an *awindow* object).

Identification of software system behaviour

An object-oriented software system can be regarded as having a group of objects together with a design which establishes a detailed order of interaction on and between objects. These interactions define the behaviour of a software system. It is helpful to define the dynamic behaviour of a software system in terms of the control flow which defines the pattern of interaction between objects. Therefore, software system behaviour focuses on when objects interact with other objects and when and how objects are created, destroyed, accessed and changed. Thus required functionality of the software system is realized as patterns of interactions between objects.

The designer should consider the sequence in which the operations must be performed. In order to identify the behaviour of a software system, three points are important:

- which operations are performed
- which attributes are manipulated
- when the operations are carried out

Answering the following questions is a useful way to characterize the behaviour of a software system:

- Which objects adequately provide particular functionality?
- When are objects created and destroyed?
- When are objects accessed and changed?
- Which request is needed in order to provide a required service?
- Which services do the objects participate in?
- When does an interaction between two objects take place?

As the design details are expanded, it is normal to realize that in order to define the behaviour of a software system and accomplish a particular service, several operations could be involved, which means that one operation of one class should invoke other operations on objects of other classes and may get some results. The pattern of these invocations establishes the control flow of the software system and shows how a particular service can be accomplished. There are basically two different trends which can be used to express the control flow of an object-oriented software system, namely centralized and decentralized control flow:

- For a centralized control flow, there is always a client object which acts as a scheduler. When a server object finishes executing an operation, the control returns to the client object before the next operation is requested.
- For a decentralized control flow, there is no unique object which can give an overall view of the activity of the software system, but the control flow is spread out among several objects. To invoke an operation, a request is sent to an object. Such an object can send other requests to other objects and the control flow passes between objects until the execution of the software system is over.

At the end of this step software system behaviour is identified in terms of interactions between objects. The definition of the behaviour of a software system is completed when the main objects in the software system have been identified and the pattern of interaction upon these objects has been defined. Operation diagrams, as shown below, can depict software system behaviour.

Operation diagrams

The functionality of a software system is defined by the services offered by that software system, and can be provided by an operation or a combination of operations. A class alone often cannot provide enough operations to meet the required functionality. Therefore, a set of operations needs to be formed so that a service can be offered. For instance, to provide the 'landing' functionality for an object of an *aeroplane* class, it is necessary to request operations to manipulate the ailerons in its right and left wings. These wings might be objects of two *leftwing* and *rightwing* subclasses of a *wing* superclass, in which the operations to control the movements of the ailerons are defined. Thus, the 'landing' functionality is provided by requesting operations of two other classes (*leftwing* and *rightwing* classes).

An operation diagram is a graphical representation showing how operations can be combined. Operation diagrams are important because they show:

- which operations are used, and when and where they are used
- relationships between operations of different classes
- which operations are needed to offer particular functionality

An example of an operation diagram is represented in Figure 8. A square represents an operation and is annotated with the operation name together with the

class name where that operation is defined. The example shows the use of an operation diagram representing the drawing of a polygon on a screen window with its possible associated label, and with the polygon then being stored in a database.

In an operation diagram, a solid arrow means that a caller operation requests a called operation and gets the control flow back after the called operation has been completed. A dashed arrow is used when a caller operation requests a called operation which retains the control flow or passes the control flow to yet another operation while the operation is being carried out. An open circle at the beginning of an arrow means a conditional passage of control flow whereas a filled circle represents an iterative passage of control flow. The numbers dictate the sequence in which the operations are requested in that operation diagram. These conventions help represent the control flow of the software system.

Because operation diagrams focus on the control flow of the software system, they also have to show information and results that might be passed between operations. Each operation has a signature which determines the parameters involved in the requesting of that operation. Parameters are split into input and output parameters. Input parameters are provided by the caller operation and can just be used by the called operation. Output parameters can only be updated by the called operation and are available to the caller operation. Parameters are represented by small arrows beside the operation which they are related to (see Figure 8). Objects can also be parameters and in this case the class to which the object belongs must appear after the identification of the object, separated by a slash. Thus, operation diagrams also enable the designer to trace the information flow.

Final remarks on MOOD

An object-oriented software system can be seen at two different layers of abstraction as shown in Figure 9. There is an upper generic layer which shows the static description of a software system and a lower level layer which consists of objects and shows the behaviour of a software system. The upper layer can represent the set of classes from which any software system may be constructed, while the lower layer depicts the actual objects

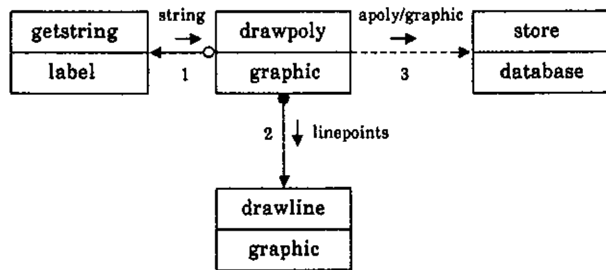

Figure 8. An operation diagram for drawing a polygon

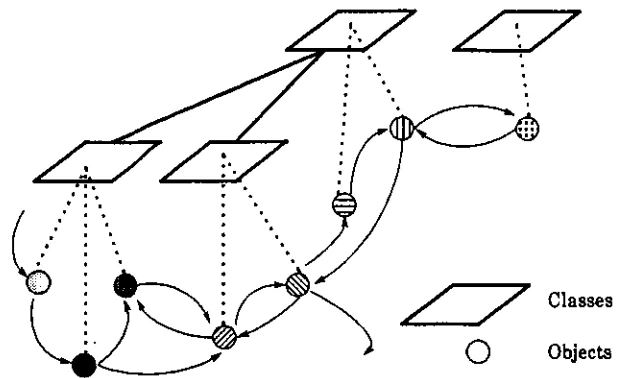

Figure 9. The static and dynamic layers of a software system

and potential object interactions required in a particular software system.

It is important to remember that identifying classes is not the same as identifying objects. Classes are a means of expressing static commonalities between objects and templates to create them, whereas objects will have a dynamic life in a running software system. The MOOD diagrams show both the static and dynamic aspects of a design. On one hand, the static design consists of class hierarchy and composition diagrams. On the other hand, the dynamic design is expanded with both object and operation diagrams which represent the overall dynamic behaviour of a software system. At first, the operations on individual objects are presented in object diagrams and then these operations are combined to perform the functionality required for the software system.

The steps proposed by MOOD, as discussed earlier in this paper, are shown in Figure 10. The steps emphasize design before implementation and are independent of any programming language. The design process originates with an abstract model of the application and culminates with a design model ready to be implemented. The path connecting these two models is not a straight line, but rather an iterative refinement process. In the course of building an object-oriented software system, the steps involved in the design of the necessary classes and objects are almost certain to be repeated many times.

MOOD has been used to design an electronic mail system, which has shown how the steps and the graphical notation introduced by the methodology can be used to produce a design of a software system in terms of classes, inheritance and object behaviour. Many problems and constraints, such as how to manipulate large numbers of classes and the difficulty of doing a complete design entirely by hand, were enough to show that the use of tools was imperative during the design phase (although the graphical notation proposed could be used manually). The limitations experimented in that design were solved afterwards by a set of tools integrated within a CASE environment which supports MOOD. This environment has also been designed using MOOD itself, as

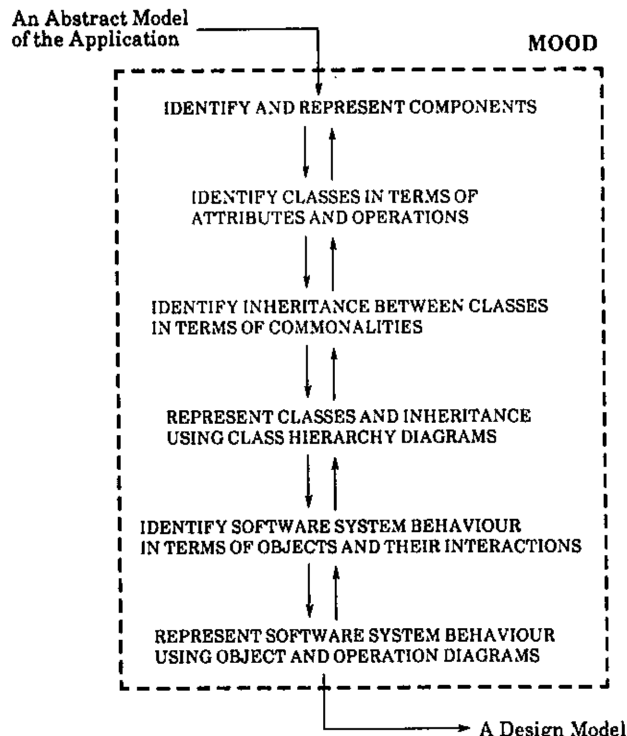

Figure 10. The steps recommended by MOOD

it can be partially seen in the figures presenting examples of MOOD diagrams throughout the paper. A prototype tool set supporting MOOD is presented in more details in Capretz[12].

In the MOOD environment there are also tools to provide consistency and completeness checks, browsers and software configuration management facilities. Moreover, although the different types of diagrams used by MOOD have been described independently, there are clearly many inter-relationships between such diagrams; for example, objects are instances of classes represented by separate class diagrams. Such interdependencies are recognized by the MOOD environment, and permit the designer to navigate around the complete set of diagrams representing an object-oriented design.

CONCLUSIONS

This paper has faced the object-oriented paradigm from a methodology standpoint, rather than from an implementation standpoint. The research has sought to establish a viable and comprehensive object-oriented design methodology which obtains the benefits of the object-oriented paradigm. There are design trends which try to integrate object-oriented concepts with different structured development methods. There also are other methodologies which lead to implementation using Ada. Nevertheless, this paper has shown that only one approach, in this case an object-oriented approach, is enough to develop software systems and it does not need to be complemented with other approaches or methodologies. This point of view encourages the designer to conform to the object-oriented paradigm and to benefit from features such as abstraction, encapsulation, inheritance and reusability, from the beginning of software development.

MOOD produces a design by progressive refinement, adding details to the same design model which is strictly object-oriented, and remains consistent through the design phase. Another result of this research has been the creation of a graphical notation to represent object-oriented design which makes use of class hierarchy diagrams, composition diagrams, object diagrams and operation diagrams. Additionally, since the use of MOOD results in an object-oriented design, it encompasses many of the benefits claimed to be inherent in any object-oriented software system, such as extensibility, modularity and clarity.

The methodology has already been applied by the authors to the design of a small number of software systems, but naturally it can evolve and mature from further experience with its application to develop large software systems. The outcome of such experience has provided a better understanding of the strengths and weaknesses of MOOD to design software systems and has also allowed evaluation and improvements to MOOD. Further experiments are expected to consolidate the proposed steps, to disseminate the notation and to boost the appearance of libraries of reusable software components for particular application domains. So far, MOOD has proved to be beneficial and has led to a better understanding of object-oriented design.

The employment of a uniform object-oriented model from system analysis to implementation facilitates a consistent binding between the various phases in software development. Therefore, it would be better if the implementation of an object-oriented design was carried out using an object-oriented programming language because concepts presented by the proposed notational conventions could be easier mapped into such a programming language. In achieving this, it becomes easier to transform one representation into another, from system analysis to implementation, and MOOD would be able to support traceability from system analysis to design and from design to implementation.

The object-oriented paradigm is such a powerful set of concepts that eventually it will get completely absorbed into the software development culture, in the same way that structured development methodologies and to some extent abstract data types concepts have been. This is evident in the abundance of research looking at various aspects of the paradigm. Consequently, the 1990s are likely to be a period of gradual acceptance of the object-oriented paradigm which will become the main approach to developing software systems in this decade. Despite its limitations, MOOD is a significant step forward in this direction.

The future of object-oriented software engineering might well be to accept a hybrid trend with other paradigms and mapping of concepts between paradigms. However, the authors believe that the object-oriented paradigm should (and hopefully will!) pervade the entire software life-cycle. The object-oriented paradigm has needed an organized and disciplined view to software development, and to be extended to cover more phases of the software life-cycle. MOOD is believed to represent an important step forward in the understanding and promotion of object-orientated design methodologies.

REFERENCES

1 **Shlaer, S and Mellor, S J** *Object-oriented systems analysis: modeling the world in data* Prentice-Hall (1988)
2 **Coad, P and Yourdon, E** *Object-oriented analysis* Prentice-Hall (1990)
3 **Booch, G** 'Object-oriented development' *IEEE Trans. Soft. Eng.* Vol 12 No 2 (February 1986) pp 211–221
4 **Seidewitz, E** 'General object-oriented software development: background and experience' *J. of Systems and Software* Vol 9 No 2 (February 1989) pp 95–108
5 **Heitz, M** *HOOD reference manual*, Issue 3.0, European Space Agency, Noordwijk, The Netherlands (1989)
6 **Jalote, P** 'Functional refinement and nested objects for object-oriented design' *IEEE Trans. Soft. Eng.* Vol 15 No 3 (March 1989) pp 264–270
7 **Booch, G** *Object-oriented design with applications* Benjamin/Cummings (1991)
8 **Wirfs-Brock, R, Wilkerson, B and Wiener, L** *Designing object-oriented software* Prentice-Hall (1990)
9 **Wasserman, A I, Pircher, P A and Muller, R J** 'The object-oriented structured design notation for software design representation' *Computer* Vol 23 No 3 (March 1990) pp 50–63
10 **Rumbaugh, J, Blaha, M, Premerlani, W, Eddy, F and Lorenson W** *Object-oriented modeling and design* Prentice-Hall (1991)
11 **Capretz, L F and Lee, P A** 'Reusability and life-cycle issues within an object-oriented methodology' in *Proc. of TOOLS-USA '92* Prentice-Hall (1992) pp 139–150
12 **Capretz, L F** *Object-oriented design methodologies for software systems* PhD Thesis, Computing Laboratory, University of Newcastle upon Tyne, UK (1991)