2

# Object-Oriented Analysis and Design

## 2.1 Introduction

This chapter surveys the most significant object-oriented design and analysis methods to emerge since the late 1980s. It concentrates primarily on OOA (Coad and Yourdon, 1991), Booch (Booch, 1991, 1994), Object Modeling Technique (Rumbaugh *et al.*, 1991), Objectory (Jacobson, 1992) and Fusion (Coleman *et al.*, 1994). It also introduces the Unified Modeling Language (Booch *et al.*, 1996; Booch and Rumbaugh, 1995).

This chapter does not aim to deal comprehensively with either the range of methods available or the fine details of each approach. Rather, it provides an overview of the design process and the strengths and weaknesses of some important and reasonably representative methods.

## 2.2 Object-Oriented Design Methods

The object-oriented design methods that we consider are all architecture-driven, incremental and iterative. They do not adopt the more traditional waterfall software development model; instead they adopt an approach which is more akin to the spiral model of Boehm (1988). This reflects developers' experiences when creating object-oriented systems – the object-oriented development process is more incremental than that for procedural systems, with less distinct barriers between analysis, design and implementation. Some organizations take this process to the extreme and adopt an evolutionary development approach. This approach delivers system functions to users in very small steps and revises project plans in the light of experience and user feedback. This philosophy has proved very successful for organizations that have fully embraced it and has led to earlier business benefits and successful end-products from large development projects.

## 2.3 Object-Oriented Analysis

We first consider the Object-Oriented Analysis approach (OOA) of Coad and Yourdon (1991). The identification of objects and classes is a crucial task in object-oriented analysis and design, but many techniques ignore this issue. For example, neither the Booch method nor OMT deal with it at all. They indicate that it is a highly creative process that can be based on the identification of nouns and verbs in an informal verbal description of the problem domain. A different approach is to use a method such as OOA as the first part of the design process and then to use another object-oriented design method for the later parts of the process.

OOA helps designers identify the detailed requirements of their software, rather than how the software should be structured or implemented. It aims to describe the existing system and how it operates, and how the software system should interact with it. One of the claims of OOA is that it helps the designer to package the requirements of the system in an appropriate manner (for object-oriented systems) and to reduce the risk of the software failing to meet the customer's requirements. In effect, OOA helps to build the Object Model that we look at in more detail when we look at OMT.

✓ There are five activities within OOA which direct the analyst during the analysis process:

- Finding classes and objects in the domain.
- Identifying structures (amongst those classes and objects). Structures are relationships such as *is-a* and *part-of.*
- Identifying subjects (related objects).
- Defining attributes (the data elements of the objects).
- Defining services (the active parts of objects that indicate what the object does).

These are not sequential steps. As information becomes available, the analyst performs the appropriate activity. The intention is that analysts can work in whatever way the domain experts find it easiest to express their knowledge. Thus, analysts may go deeper into one activity than the others as the domain experts provide greater information in that area. Equally, analysts may jump around between activities, identifying classes one minute and services the next.

### ✓ 2.3.1 Class Responsibility Collaborator (CRC)

CRC (Class Responsibility Collaborator) is an exploratory technique rather than a complete method. It was originally devised as a way of teaching basic concepts in object-oriented design. The CRC technique can be exploited in other methods (for example, Booch; it is explicitly used as an early step in Fusion). It is also the foundation of the responsibility-driven design method (Wirfs-Brock *et al.*, 1990), where it constitutes the first phase.

CRC deals primarily with the design phase of development. The process is anthropomorphic and drives development by having project teams enact scenarios and play the parts of objects. Classes are recorded on index cards. The steps in the process can be summarized in Figure 2.1.

#### Identification of Classes and Responsibilities

In this stage, the classes are identified. Guidelines include looking for nouns in the requirements document and modelling physical objects and conceptual entities. Object categories are candi-

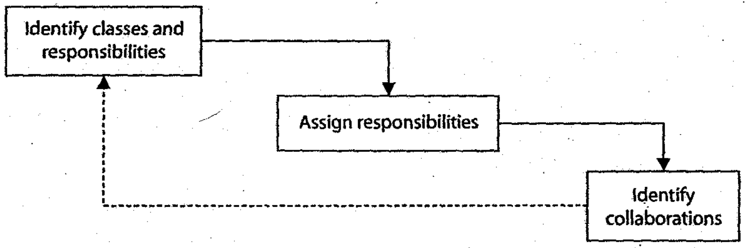

**Figure 2.1** Steps in CRC.

date classes. Grouping classes by common attributes gives candidates for abstract superclasses. The classes are written on cards. The classes form the vocabulary for discussion. Subclasses and superclasses are also identified and recorded on the class card. The requirements are then examined for actions and information associated with each class to find the responsibilities of the classes. *Responsibilities* are the essential duties that have to be performed. They identify problems to be solved and are a handle for discussing solutions. An example of a CRC class card is presented in Figure 2.2.

### Assignment of Responsibilities

The responsibilities identified in the previous stage are allocated to classes. The goal is to distribute the "intelligence" of the system evenly around the classes, with behaviour kept with related information. Information about one thing should appear in just one place. If necessary, responsibilities can be shared among related classes. Responsibilities should be made as general as possible and placed as high as possible in the inheritance hierarchy.

### Identification of Collaborations

This stage identifies how classes interact. Each class/responsibility pair is examined to see which other classes would need to be consulted to fulfil the responsibility and which classes make use of which responsibilities. The cards for classes that closely collaborate are grouped together physically. This informal grouping helps in the understanding of the emerging design.

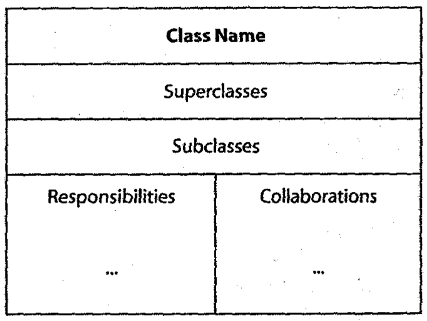

**Figure 2.2** A CRC class card.

Refinement

The design process is driven toward completion by considering execution scenarios. Each member of the design team takes the part of a class enacting the scenario. This process uncovers missing responsibilities and collaborators. The restricted size of the index cards helps stop classes becoming too complex. If a card becomes too cluttered, then it is reviewed. The outcome can be simplified statements of responsibilities, new subclasses or superclasses, or even new classes.

The output of the design is a set of classes that are related through inheritance. The hierarchy is refined with common responsibilities placed as high as possible in the graph. Abstract classes cannot inherit from concrete ones and classes that add no functionality can be discarded.

#### *Strengths and Weaknesses*

In assessing CRC it should be noted that CRC is a technique and does not claim to be a method (even though there are practitioners around who use it as a method). CRC is primarily an exploratory technique can be very useful in this role. However, it does not produce a design which can be directly implemented – it is at too high a level. As the index cards produced during the design process are the only form of documentation associated with the design (i.e. the only record of the design decisions taken as well as the end result) they are clearly inadequate. It can be a very powerful technique for identifying initial classes and class relationships (both inheritance and uses relations); however, it does not deal with object creation, object partitioning, object interactions (only collaborations) or any issues related to the implementation of the system. In summary CRC is a powerful technique which has its place within other design methods during the very early stages of class and object identification.

## ✓ 2.4 The Booch Method

The Booch method (also known as Booch and Object-oriented Development, or OOD) is one of the earliest recognizable object-oriented design methods. It was first described in a paper published in 1986 and has become widely adopted since the publication of a book describing the method (Booch, 1991, 1994).

The Booch method provides a step-by-step guide to the design of an object-oriented system. Although Booch's books discuss the analysis phase, they do so in too little detail compared with the design phase.

### 2.4.1 The Steps in the Booch Method

- *Identification of classes and objects* involves analyzing the problem domain and the system requirements to identify the set of classes required. This is not trivial and relies on a suitable requirements analysis.
- *Identification of the semantics of classes and objects* involves identifying the services offered by an object and required by an object. A service is a function performed by an object, and during this step the overall system functionality is devolved among the objects. This is

another non-trivial step, and it may result in modifications to the classes and objects identified in the last step.
- *Identification of the relationships between classes and objects* involves identifying links between objects as well and inheritance between classes. This step may identify new services required of objects.
- *Implementation of classes and objects* attempts to consider how to implement the classes and objects and how to define the attributes and provide services. This involves considering algorithms. This process may lead to modifications in the deliverables of all of the above steps and may force the designer to return to some or all of the above steps.

During these steps, the designer produces

- Class diagrams, which illustrate the classes in the system and their relationships.
- Object diagrams, which illustrate the actual objects in the system and their relationships.
- Module diagrams, which package the classes and objects into modules. These modules illustrate the influence that Ada had on the development of the Booch method (Booch, 1987).
- Process diagrams, which package processes and processors.
- State transition diagrams and timing diagrams, which describe the dynamic behaviour of the system (the other diagrams describe the static structure of the system).

Booch recommends an incremental and iterative development of a system through the refinement of different yet consistent logical and physical views of that system.

### 2.4.2 Strengths and Weaknesses

The biggest problem for a designer approaching the Booch method for the first time is that the plethora of different notations is supported by a poorly defined and loose process (although the revision to the method described in Booch (1994) addresses this to some extent). It does not give step-by-step guidance and possesses very few mechanisms for determining the system's requirements. Its main strengths are its (mainly graphical) notations, which cover most aspects of the design of an object-oriented system, and its greatest weakness is the lack of sufficient guidance in the generation of these diagrams.

## 2.5 The Object Modeling Technique

The Object Modeling Technique (OMT) is an object-oriented design method which aims to construct a series of models which refine the system design until the final model is suitable for implementation. The design process is divided into three phases:

- The Analysis Phase attempts to model the problem domain.
- The Design Phase structures the results of the analysis phase in an appropriate manner.
- The Implementation Phase takes into account target language constructs.

### 2.5.1 The Analysis Phase

Three types of model are produced by the analysis phase:

- *The object model* represents the static structure of the domain. It describes the objects, their classes and the relationships between the objects. For example, the object model might represent the fact that a department object possesses a single manager (object) but many employees (objects). The notation is based on an extension of the basic entity-relationship notation.
- *The dynamic model* represents the behaviour of the system. It expresses what happens in the domain, when it occurs and what effect it has. It does not represent how the behaviour is achieved. The formalism used to express the dynamic model is based on a variation of finite state machines called statecharts. These were developed by Harel and others (Harel *et al.*, 1987; Harel, 1988) to represent dynamic behaviour in real-time avionic control systems. Statecharts indicate the states of the system, the transitions between states, their sequence and the events which cause the state change.
- *The functional model* describes how system functions are performed. It uses data flow diagrams which illustrate the sources and sinks of data as well as the data being exchanged. They contain no sequencing information or control structures.

The relationship between these three models is important, as each model adds to the designer's understanding of the domain:

- The object model defines the objects which hold the state variables referenced in the dynamic model and are the sources and sinks referenced in the functional model.
- The dynamic model indicates when the behaviour in the functional model occurs and what triggers it.
- The functional model explains why an event transition leads from one state to another in the dynamic model.

You do not build these models sequentially; changes to any one of the models may have a knock-on effect in the other models. Typically, the designer starts with the object model, then considers the dynamic model and finally the functional model, but the process is iterative.

The analysis process is described in considerable detail and provides step-by-step guidance. This ensures that the developer knows what to do at any time to advance the three models.

### 2.5.2 The Design Phase

The design phase of OMT builds upon the models produced during the analysis phase:

- *The system design step* breaks the system down into subsystems and determines the overall architecture to be used.
- *The object design step* decides on the algorithms to be used for the methods. The methods are identified by examining the three analysis models for each class, etc.

Each of the steps gives some guidelines for their respective tasks; however, far less support is provided for the designer than in the analysis phase. For example, there is no systematic guidance for the identification of subsystems, although the issues involved are discussed (resource management, batch versus interactive modes etc.). This means that it can be difficult to identify where to start, how to proceed and what to do next.

### 2.5.3 The Implementation Phase

The implementation phase codifies the system and object designs into the target language. This phase provides some very useful information on how to implement features used in the model-based design process used, but it lacks the step-by-step guidance which would be useful for those new to object orientation.

### 2.5.4 Strengths and Weaknesses

OMT's greatest strength is the level of step-by-step support that it provides during the analysis phase. However, it is much weaker in its guidance during the design and implementation phases, where it provides general guidelines (and some heuristics).

## 2.6 The Objectory Method

The driving force behind the Objectory method (Jacobson *et al.*, 1992) is the concept of a *use case*. A use case is a particular interaction between the system and a user of that system (an actor) for a particular purpose (or function). The users of the system may be human or machine. A complete set of use cases therefore describes a system's functionality based around what actors should be able to do with the system. The Objectory method has three phases, which produce a set of models.

### 2.6.1 The Requirements Phase

The requirements phase uses a natural language description of what the system should do to build three models.

- *The use case model* describes the interactions between actors and the system. Each use case specifies the actions that are performed and their sequence. Any alternatives are also documented. This can be done in natural language or using state transition diagrams.
- *The domain model* describes the objects, classes and associations between objects in the domain. It uses a modified entity–relationship model.
- *The user interface descriptions* contain mock-ups of the various interfaces between actors and the system. User interfaces are represented as pictures of windows, while other interfaces are described by protocols.

### 2.6.2 The Analysis Phase

The analysis phase produces the analysis model and a set of subsystem descriptions. The analysis model is a refinement of the domain object model produced in the requirements phase. It contains behavioural information as well as control objects which are linked to use cases. The analysis model also possesses entity objects (which exist beyond a single use case) and interface objects (which handle system–actor interaction). The subsystem descriptions partition the system around objects which are involved in similar activities and which are closely coupled. This organization structures the rest of the design process.

### 2.6.3 The Construction Phase

The construction phase refines the models produced in the analysis phase. For example, inter-object communication is refined and the facilities provided by the target language are considered. This phase produces three models:

- Block models represent the functional modules of the system.
- Block interfaces specify the public operations performed by blocks.
- Block specifications are optional descriptions of block behaviour in the form of finite state machines.

The final stage is to implement the blocks in the target language.

### 2.6.4 Strengths and Weaknesses

The most significant aspect of Objectory is its use of use cases, which join the building blocks of the method. Objectory is unique among the methods considered here, as it provides a unifying framework for the design process. However, it still lacks the step-by-step support which would simplify the whole design process.

## 2.7 The Fusion Method

The majority of object-oriented design methods currently available, including those described in this chapter, take a systematic approach to the design process. However, in almost all cases this process is rather weak, providing insufficient direction or support to the developer. In addition, methods such as OMT rely on a "bottom up" approach. This means that developers must focus on the identification of appropriate classes and their interfaces without necessarily having the information to enable them to do this in an appropriate manner for the overall system. Little reference is made to the system's overall functionality when determining class functionality etc. Indeed, some methods provide little more than some vague guidelines and anecdotal heuristics.

In contrast, Fusion explicitly attempts to provide a systematic approach to object-oriented software development. In many ways, the Fusion method is a mixture of a range of other approaches (indeed, the authors of the method acknowledge that there is little new in

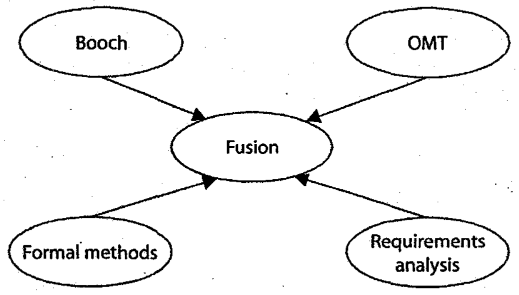

**Figure 2.3** Some of the influences on Fusion.

the approach, other than that they have put it all together in a single method; see Figure 2.3).

As with other object-oriented design methods, Fusion is based around the construction of appropriate models that capture different elements of the system and different knowledge. These models are built up during three distinct phases:

- *The analysis phase* produces models that describe the high-level constraints from which the design models are developed.
- *The design phase* produces a set of models that describe how the system behaves in terms of a collection of interacting objects.
- *The implementation phase* describes how to map the design models onto implementation language constructs.

Within each phase a set of detailed steps attempts to guide the developer through the Fusion process. These steps include checks to ensure the consistency and completeness of the emerging design. In addition, the output of one step acts as the input for the next.

Fusion's greatest weakness is its complexity – it really requires a sophisticated CASE tool. Without such a tool, it is almost impossible to produce a consistent and complete design.

## 2.8 The Unified Modeling Language

The Unified Modeling Language (UML) is an attempt by Grady Booch, Ivar Jacobson and James Rumbaugh to build on the experiences of the Booch, Object Modeling Technique (OMT) and Objectory methods. Their aim is to produce a single, common, and widely useable modelling language for these methods and, working with other methodologists, for other methods. This means that UML focuses on a standard language and not a standard process, which reflects what happens in reality; a particular notation is adopted as the means of communication on a specific project and between projects. However, between projects (and sometimes within projects), different design methods are adopted as appropriate. For example, a design method intended for the domain of real-time avionics systems may not be suitable for designing a small payroll system. The UML is an attempt to develop a common meta-model which unifies semantics and from which a common notation can be built.

## 2.9 Summary

In this chapter, we have reviewed a number of object-oriented analysis and design methods and the Unified Modeling Language. We have briefly considered the features, strengths and weaknesses of each method.

In all these systems, during the design process it is often difficult to identify commonalities between classes at the implementation level. This means that, during the implementation phase, experienced object-oriented technicians should look for situations in which they can move implementation-level components up the class hierarchy. This can greatly increase the amount of reuse within a software system and may lead to the introduction of abstract classes that contain the common code.

The problem with this is that the implemented class hierarchy no longer reflects the design class hierarchy. It is therefore necessary to have a free flow of information between the implementation and design phases in an object-oriented project.

## 2.10 References

Boehm, B.W. (1988). A spiral model of software development and enhancement. *IEEE Computer*, May, pp. 61–72.

Booch, G. (1987). *Software Components with Ada*. Benjamin Cummings, Menlo Park, CA.

Booch, G. (1991). *Object-Oriented Design with Applications*. Benjamin Cummings, Redwood City, CA.

Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*, 2nd edn. Benjamin Cummings, Redwood City, CA.

Booch, G. and Rumbaugh, J. (1995). *The Unified Method Documentation Set*, Version 0.8. Rational Software Corporation (available at `http://www.rational.com/ot/uml.html/`).

Booch, G., Jacobson, I. and Rumbaugh, J. (1996). *The Unified Modeling Language for Object-oriented Development, Documentation Set*, Version 0.91 Addendum, UML Update. Rational Software Corporation (available at `http://www.rational.com/ot/uml.html/`).

Coad, P. and Yourdon, E. (1991). *Object-Oriented Analysis*. Yourdon Press, Englewood Cliffs, NJ.

Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F. and Jeremes, P. (1994). *Object-oriented Development: The Fusion Method*. Prentice Hall, Englewood Cliffs, NJ.

Harel, D. (1988). On visual formalisms. *Communications of the ACM*, 31(5), 514–30.

Harel, D. *et al.* (1987). On the formal semantics of Statecharts. *Proceedings of the 2nd IEEE Symposium on Logic in Computer Science*, pp. 54–64.

Jacobson, I. *et al.* (1992). *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, MA.

Rumbaugh, J. *et al.* (1991). *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, NJ.

Wirfs-Brock, R., Wilkerson, B. and Wiener, W. (1990). *Designing Object-Oriented Software.* Prentice Hall, Englewood Cliffs, NJ.