PART I

Overview

1

Introduction

1.1 SOFTWARE MODELING

Modeling is used in many walks of life, going back to early civilizations such as Ancient Egypt, Rome, and Greece, where modeling was used to provide small-scale plans in art and architecture (Figure 1.1). Modeling is widely used in science and engineering to provide abstractions of a system at some level of precision and detail. The model is then analyzed in order to obtain a better understanding of the system being developed. According to the Object Modeling Group (OMG), "modeling is the designing of software applications before coding."

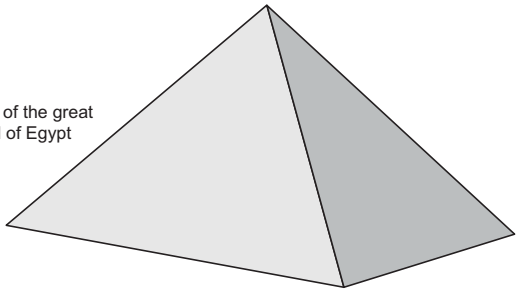
In model-based software design and development, software modeling is used as an essential part of the software development process. Models are built and analyzed prior to the implementation of the system, and are used to direct the subsequent implementation.

A better understanding of a system can be obtained by considering it from different perspectives (also referred to as multiple views) (Gomaa 2006; Gomaa and Shin 2004), such as requirements models, static models, and dynamic models of the software system. A graphical modeling language such as UML helps in developing, understanding, and communicating the different views.

This chapter introduces object-oriented methods and notations, an overview of software modeling and architectural design, and an introduction to model-driven architecture and UML. The chapter then briefly describes the evolution of software design methods, object-oriented analysis and design methods, and concurrent, distributed, and real-time design methods.

1.2 OBJECT-ORIENTED METHODS AND THE UNIFIED MODELING LANGUAGE

Object-oriented concepts are crucial in software analysis and design because they address fundamental issues of software modifiability, adaptation, and evolution. Object-oriented methods are based on the concepts of information hiding, classes, and inheritance. Information hiding can lead to systems that are more self-contained



a) A model of the great pyramid of Egypt

b) The great pyramid of Egypt

Figure 1.1. Example of modeling and architecture

and hence are more modifiable and maintainable. Inheritance provides an approach for adapting a class in a systematic way.

With the proliferation of notations and methods for the object-oriented analysis and design of software applications, the Unified Modeling Language (UML) was developed to provide a standardized graphical language and notation for describing object-oriented models. However, because UML is methodology-independent, it needs to be used together with an object-oriented analysis and design method. Because the UML is now the standardized graphical language and notation for describing object-oriented models, this book uses the UML notation throughout.

Modern object-oriented analysis and design methods are model-based and use a combination of use case modeling, static modeling, state machine modeling, and object interaction modeling. Almost all modern object-oriented methods use the UML notation for describing software requirements, analysis, and design models (Booch, Rumbaugh, and Jacobson 2005; Fowler 2004; Rumbaugh, Booch, and Jacobson 2005).

In **use case modeling**, the functional requirements of the system are defined in terms of use cases and actors. **Static modeling** provides a structural view of the system. Classes are defined in terms of their attributes, as well as their relationships with other classes. **Dynamic modeling** provides a behavioral view of the system. The use cases are realized to show the interaction among participating objects. Object interaction diagrams are developed to show how objects communicate with each other to realize the use case. The state-dependent aspects of the system are defined with statecharts.

1.3 SOFTWARE ARCHITECTURAL DESIGN

A **software architecture** (Bass, Clements, and Kazman 2003; Shaw and Garlan 1996) separates the overall structure of the system, in terms of components and their interconnections, from the internal details of the individual components. The emphasis on components and their interconnections is sometimes referred to as *programming-in-the-large*, and the detailed design of individual components is referred to as *programming-in-the-small.*

A software architecture can be described at different levels of detail. At a high level, it can describe the decomposition of the software system into subsystems. At a lower level, it can describe the decomposition of subsystems into modules or components. In each case, the emphasis is on the external view of the subsystem/component – that is, the interfaces it provides and requires – and its interconnections with other subsystems/components.

The software quality attributes of a system should be considered when developing the software architecture. These attributes relate to how the architecture addresses important nonfunctional requirements, such as performance, security, and maintainability.

The software architecture is sometimes referred to as a high-level design. A software architecture can be described from different views, as described in Section 1.7. It is important to ensure that the architecture fulfills the software requirements, both functional (what the software has to do) and nonfunctional (how well it should do it). It is also the starting point for the detailed design and implementation, when typically the development team becomes much larger.

1.4 METHOD AND NOTATION

This section defines important terms for software design.

A **software design notation** is a means of describing a software design either graphically or textually, or both. For example, class diagrams are a graphical design notation, and pseudocode is a textual design notation. UML is a graphical notation for object-oriented software applications. A design notation suggests a particular approach for performing a design; however, it does not provide a systematic approach for producing a design.

A **software design concept** is a fundamental idea that can be applied to designing a system. For example, information hiding is a software design concept.

A **software design strategy** is an overall plan and direction for developing a design. For example, object-oriented decomposition is a software design strategy.

Software structuring criteria are heuristics or guidelines used to help a designer in structuring a software system into its components. For example, object structuring criteria provide guidelines for decomposing the system into objects.

A **software design method** is a systematic approach that describes the sequence of steps to follow in order to create a design, given the software requirements of the application. It helps a designer or design team identify the design decisions to be made, the order in which to make them, and the structuring criteria to use in making them. A design method is based on a set of design concepts, employs one or more design strategies, and documents the resulting design, using a design notation.

During a given design step, the method might provide a set of structuring criteria to help the designer in decomposing the system into its components.

The Collaborative Object Modeling and Design Method, or COMET, uses the UML notation to describe the design. COMET is based on the design concepts of information hiding, classes, inheritance, and concurrent tasks. It uses a design strategy of concurrent object design, which addresses the structuring of a software system into active and passive objects and defines the interfaces between them. It provides structuring criteria to help structure the system into objects during analysis, and additional criteria to determine the subsystems and concurrent tasks during design.

1.5 COMET: A UML-BASED SOFTWARE MODELING AND DESIGN METHOD FOR SOFTWARE APPLICATIONS

This book describes a UML-based software modeling and architectural design method called COMET. COMET is an iterative use case driven and object-oriented software development method that addresses the requirements, analysis, and design modeling phases of the software development life cycle. The functional requirements of the system are defined in terms of actors and use cases. Each use case defines a sequence of interactions between one or more actors and the system. A use case can be viewed at various levels of detail. In a *requirements* model, the functional requirements of the system are defined in terms of actors and use cases. In an *analysis* model, the use case is realized to describe the objects that participate in the use case and their interactions. In the *design* model, the software architecture is developed, addressing issues of distribution, concurrency, and information hiding.

1.6 UML AS A STANDARD

This section briefly reviews the evolution of UML into a standard modeling language and notation for describing object-oriented designs. The evolution of UML is described in detail by Kobryn (1999). UML 0.9 unified the modeling notations of Booch, Jacobson (1992), and Rumbaugh et al. (1991). This version formed the basis of a standardization effort, with the additional involvement of a diverse mix of vendors and system integrators. The standardization effort culminated in submission of the initial UML 1.0 proposal to the OMG in January 1997. After some revisions, the final UML 1.1 proposal was submitted later that year and adopted as an object modeling standard in November 1997.

The OMG maintains UML as a standard. The first adopted version of the standard was UML 1.3. There were minor revisions with UML 1.4 and 1.5. A major revision to the notation was made in 2003 with UML 2.0. The latest books on UML conform to UML 2.0, including the revised editions of Booch, Rumbaugh, and Jacobson (2005), Rumbaugh, Booch, and Jacobson (2005), Fowler (2004), Eriksson et al. (2004), and Douglass (2004). There have been minor revisions since then. The current version of the standard is referred to as UML 2.

1.6.1 Model-Driven Architecture with UML

In the OMG's view, "modeling is the designing of software applications before coding." The OMG promotes model-driven architecture as the approach in which UML models of the software architecture are developed prior to implementation. According to the OMG, UML is methodology-independent; UML is a notation for describing the results of an object-oriented analysis and design developed via the methodology of choice.

A UML model can be either a platform-independent model (PIM) or a platform-specific model (PSM). The PIM is a precise model of the software architecture before a commitment is made to a specific platform. Developing the PIM first is particularly useful because the same PIM can be mapped to different middleware platforms, such as COM, CORBA, .NET, J2EE, Web Services, or another Web platform. The approach in this book is to use the concept of model-driven architecture to develop a component-based software architecture, which is expressed as a UML platform–independent model.

1.7 MULTIPLE VIEWS OF SOFTWARE ARCHITECTURE

A software architecture can be considered from different perspectives, which are referred to as different views. Kruchten (Kruchten 1995) introduced the 4+1 view model of software architecture, in which he advocated a multiple-view modeling approach for software architectures, in which the use case view is the unifying view (the 1 view of the 4+1 views). The views are the logical view, which is a static modeling view; the process view, which is a concurrent process or task view; and the development view, which is a subsystem and component design view. Hofmeister et al. (2000) describe an industrial perspective on applied software architecture consisting of four views: a conceptual view, which describes the main design elements and the relationships between them; a code view, which consists of the source code organized into object code, libraries, and directories; a module view, which consists of subsystems and modules; and an execution view, which is a concurrent and distributed execution perspective.

In this book, we will describe and depict the different modeling views of the software architecture in UML. The views are as follows:

- **Use case view.** This view is a functional requirements view, which is an input to develop the software architecture. Each use case describes the sequence of interactions between one or more actors (external users) and the system.
- **Static view.** The architecture is depicted in terms of classes and relationships, which can be associations, whole/part relationships (compositions or aggregations), or generalization/specialization relationships. Depicted on UML class diagrams.
- **Dynamic interaction view.** This view describes the architecture in terms of objects as well as the message communication between them. This view can also be used to depict the execution sequence of specific scenarios. Depicted on UML communication diagrams.

- **Dynamic state machine view.** The internal control and sequencing of a control component can be depicted using a state machine. Depicted on UML statechart diagrams.
- **Structural component view.** The software architecture is depicted in terms of components, which are interconnected through their ports, which in turn support provided and required interfaces. Depicted on UML structured class diagrams.
- **Dynamic concurrent view.** The software architecture is viewed as concurrent components, executing on distributed nodes, and communicating by messages. Depicted on UML concurrent communication diagrams.
- **Deployment view.** This depicts a specific configuration of the distributed architecture with components assigned to hardware nodes. Depicted on UML deployment diagrams.

1.8 EVOLUTION OF SOFTWARE MODELING AND DESIGN METHODS

In the 1960s, programs were often implemented with little or no systematic requirements analysis and design. Graphical notations – in particular, flowcharts – were often used, either as a documentation tool or as a design tool for planning a detailed design prior to coding. Subroutines were originally created as a means of allowing a block of code to be shared by calling it from different parts of a program. They were soon recognized as a means of constructing modular systems and were adopted as a project management tool. A program could be divided up into modules, where each module could be developed by a separate person and implemented as a subroutine or function.

With the growth of structured programming in the early seventies, the ideas of top-down design and stepwise refinement (Dahl 1972) gained prominence as program design methods, with the goal of providing a systematic approach for structured program design. Dijkstra developed one of the first software design methods with the design of the T.H.E. operating system (Dijkstra 1968), which used a hierarchical architecture. This was the first design method to address the design of a concurrent system, namely, an operating system.

In the mid- to late 1970s, two different software design strategies gained prominence: data flow–oriented design and data structured design. The data flow oriented–design approach as used in Structured Design (see Budgen [2003] for an overview) was one of the first comprehensive and well-documented design methods to emerge. The idea was that a better understanding of the functions of the system could be obtained by considering the flow of data through the system. It provided a systematic approach for developing data flow diagrams for a system and then mapping them to structure charts. Structured Design introduced the coupling and cohesion criteria for evaluating the quality of a design. This approach emphasized functional decomposition into modules and the definition of module interfaces. The first part of Structured Design, based on data flow diagram development, was refined and extended to become a comprehensive analysis method, namely, Structured Analysis (see Budgen [2003] for an overview).

An alternative software design approach was that of data structured design. This view was that a full understanding of the problem structure is best obtained from consideration of the data structures. Thus, the emphasis is on first designing the

data structures and then designing the program structures based on the data structures. The two principal design methods to use this strategy were Jackson Structured Programming (Jackson 1983) and the Warnier/Orr method.

In the database world, the concept of separating logical and physical data was key to the development of database management systems. Various approaches were advocated for the logical design of databases, including the introduction of entity-relationship modeling by Chen.

Parnas (1972) made a great contribution to software design with his advocacy of information hiding. A major problem with early systems, even in many of those designed to be modular, resulted from the widespread use of global data, which made these systems prone to error and difficult to change. Information hiding provided an approach for greatly reducing, if not eliminating, global data.

A major contribution for the design of concurrent and real-time systems came in the late 1970s with the introduction of the MASCOT notation and later the MASCOT design method. Based on a data flow approach, MASCOT formalized the way tasks communicate with each other, either through channels for message communication or through pools (information-hiding modules that encapsulate shared data structures). The data maintained by a channel or pool are accessed by a task only indirectly by calling access procedures provided by the channel or pool. The access procedures also synchronize access to the data, typically using semaphores, so that all synchronization issues are hidden from the calling task.

There was a general maturation of software design methods in the 1980s, and several system design methods were introduced. Parnas's work with the Naval Research Lab (NRL), in which he explored the use of information hiding in large-scale software design, led to the development of the Naval Research Lab Software Cost Reduction Method (Parnas, Clements, and Weiss 1984). Work on applying Structured Analysis and Structured Design to concurrent and real-time systems led to the development of Real-Time Structured Analysis and Design (RTSAD) (see Gomaa [1993] for an overview) and the Design Approach for Real-Time Systems (DARTS) (Gomaa 1984) methods.

Another software development method to emerge in the early 1980s was Jackson System Development (JSD) (Jackson 1983). JSD views a design as being a simulation of the real world and emphasizes modeling entities in the problem domain by using concurrent tasks. JSD was one of the first methods to advocate that the design should model reality first and, in this respect, predated the object-oriented analysis methods. The system is considered a simulation of the real world and is designed as a network of concurrent tasks, in which each real-world entity is modeled by means of a concurrent task. JSD also defied the then-conventional thinking of top-down design by advocating a middle-out behavioral approach to software design. This approach was a precursor of object interaction modeling, an essential aspect of modern object-oriented development.

1.9 EVOLUTION OF OBJECT-ORIENTED ANALYSIS AND DESIGN METHODS

In the mid- to late 1980s, the popularity and success of object-oriented programming led to the emergence of several object-oriented design methods, including

Booch, Wirfs-Brock, Wilkerson, and Wiener (1990), Rumbaugh et al. (1991), Shlaer and Mellor (1988, 1992), and Coad and Yourdon (1991, 1992). The emphasis in these methods was on modeling the problem domain, information hiding, and inheritance.

Parnas advocated using information hiding as a way to design modules that were more self-contained and therefore could be changed with little or no impact on other modules. Booch introduced object-oriented concepts into design initially with information hiding, in the **object-based design** of Ada-based systems and later extended this to using information hiding, classes, and inheritance in **object-oriented design**. Shlaer and Mellor (1988), Coad and Yourdon (1991), and others introduced object-oriented concepts into analysis. It is generally considered that the object-oriented approach provides a smoother transition from analysis to design than the functional approach.

Object-oriented analysis methods apply object-oriented concepts to the analysis phase of the software life cycle. The emphasis is on identifying real-world objects in the problem domain and mapping them to software objects. The initial attempt at object modeling was a static modeling approach that had its origins in information modeling, in particular, entity-relationship (E-R) modeling or, more generally, semantic data modeling, as used in logical database design. Entities in E-R modeling are information-intensive objects in the problem domain. The entities, the attributes of each entity, and relationships among the entities, are determined and depicted on E-R diagrams; the emphasis is entirely on data modeling. During design, the E-R model is mapped to a database, usually relational. In object-oriented analysis, objects in the problem domain are identified and modeled as software classes, and the attributes of each class, as well as the relationships among classes, are determined (Coad 1991; Rumbaugh et al. 1991; Shlaer and Mellor 1988).

The main difference between *classes* in static object-oriented modeling and *entity types* in E-R modeling is that classes have operations but entity types do not have operations. In addition, whereas information modeling only models persistent entities that are to be stored in a database, other problem domain classes are also modeled in static object modeling. The advanced information modeling concepts of aggregation and generalization/specialization are also used. The most widely used notation for static object modeling before UML was the Object Modeling Technique (OMT) (Rumbaugh et al. 1991).

Static object modeling was also referred to as *class modeling* and *object modeling* because it involves determining the classes to which objects belong and depicting classes and their relationships on class diagrams. The term *domain modeling* is also used to refer to static modeling of the problem domain (Rosenberg and Scott 1999; Shlaer and Mellor 1992).

The early object-oriented analysis and design methods emphasized the structural aspects of software development through information hiding and inheritance but neglected the dynamic aspects. A major contribution by the OMT (Rumbaugh et al. 1991) was to clearly demonstrate that dynamic modeling was equally important. In addition to introducing the static modeling notation for the object diagrams, OMT showed how dynamic modeling could be performed with statecharts for showing the state-dependent behavior of active objects and with sequence diagrams to show the sequence of interactions between objects. Rumbaugh et al. (1991) used statecharts,

which are hierarchical state transition diagrams originally conceived by Harel (1988, 1998), for modeling active objects. Shlaer and Mellor (1992) also used state transition diagrams for modeling active objects. Booch initially used object diagrams to show the instance-level interactions among objects and later sequentially numbered the interactions to more clearly depict the communication among objects.

Jacobson (1992) introduced the use case concept for modeling the system's functional requirements. Jacobson also used the sequence diagram to describe the sequence of interactions between the objects that participate in a use case. The use case concept was fundamental to all phases of Jacobson's object-oriented software engineering life cycle. The use case concept has had a profound impact on modern object-oriented software development.

Prior to UML, there were earlier attempts to unify the various object-oriented methods and notations, including Fusion (Coleman et al. 1993) and the work of Texel and Williams (1997). The UML notation was originally developed by Booch, Jacobson, and Rumbaugh to integrate the notations for use case modeling, static modeling, and dynamic modeling (using statecharts and object interaction modeling), as described in Chapter 2. Other methodologists also contributed to the development of UML. An interesting discussion of how UML has evolved and how it is likely to evolve in the future is given in Cobryn [1999] and Selic (1999).

1.10 SURVEY OF CONCURRENT, DISTRIBUTED, AND REAL-TIME DESIGN METHODS

The Concurrent Design Approach for Real-Time Systems (CODARTS) method (Gomaa 1993) built on the strengths of earlier concurrent design, real-time design, and early object-oriented design methods by emphasizing both information-hiding module structuring and concurrent task structuring.

Octopus (Awad, Kuusela, and Ziegler) is a real-time design method based on use cases, static modeling, object interactions, and statecharts. For real-time systems, ROOM (Selic, Gullekson, and Ward 1994) is an object-oriented real-time design method that is closely tied in with a **CASE** (Computer-Assisted Software Engineering) tool called ObjecTime. ROOM is based around actors, which are active objects that are modeled using a variation on statecharts called ROOMcharts. A ROOM model is capable of being executed and thus could be used as an early prototype of the system.

Buhr (1996) introduced an interesting concept called the use case map (based on the use case concept) to address the issue of dynamic modeling of large-scale systems.

For UML-based real-time software development, Douglass (2004, 1999) has provided a comprehensive description of how UML can be applied to real-time systems.

An earlier version of the COMET method for designing concurrent, real-time, and distributed applications, which is based on UML 1.3, is described in Gomaa (2000). This new textbook expands on the COMET method by basing it on UML 2, increasing the emphasis on software architecture, and addressing a wide range of software applications, including object-oriented software architectures, client/server software architectures, service-oriented architectures, component-based software