Introduction

Object-oriented modeling and design is a way of thinking about problems using models organized around real-world concepts. The fundamental construct is the object, which combines both data structure and behavior. Object-oriented models are useful for understanding problems, communicating with application experts, modeling enterprises, preparing documentation, and designing programs and databases. This book presents an object-oriented notation and process that extends from analysis through design to implementation. The same notation applies at all stages of the process as development proceeds.

1.1 What Is Object-Orientation?

Superficially the term object-oriented (OO) means that we organize software as a collection of discrete objects that incorporate both data structure and behavior. This contrasts with previous programming approaches in which data structure and behavior are only loosely connected. There is some dispute about exactly what characteristics are required by an OO approach, but they generally include four aspects: identity, classification, inheritance, and polymorphism.

Identity means that data is quantized into discrete, distinguishable entities called objects. The first paragraph in this chapter, my workstation, and the white queen in a chess game are examples of objects. Figure 1.1 shows some additional objects. Objects can be concrete, such as a file in a file system, or conceptual, such as a scheduling policy in a multiprocess-  

cessing operating system. Each object has its own inherent identity. In other words, two objects are distinct even if all their attribute values (such as name and size) are identical.

In the real world an object simply exists, but within a programming language each object has a unique handle by which it can be referenced. Languages implement the handle in various ways, such as an address, array index, or artificial number. Such object references are uniform and independent of the contents of the objects, permitting mixed collections of objects to be created, such as a file system directory that contains both files and subdirectories.
Classification means that objects with the same data structure (attributes) and behavior (operations) are grouped into a class. Paragraph, Monitor, and ChessPiece are examples of classes. A class is an abstraction that describes properties important to an application and ignores the rest. Any choice of classes is arbitrary and depends on the application.

Each class describes a possibly infinite set of individual objects. Each object is said to be an instance of its class. An object has its own value for each attribute but shares the attribute names and operations with other instances of the class. Figure 1.2 shows two classes and some of their respective instances. An object contains an implicit reference to its own class; it “knows what kind of thing it is.”

Inheritance is the sharing of attributes and operations (features) among classes based on a hierarchical relationship. A superclass has general information that subclasses refine and elaborate. Each subclass incorporates, or inherits, all the features of its superclass and adds its own unique features. Subclasses need not repeat the features of the superclass. For example, ScrollingWindow and FixedWindow are subclasses of Window. Both subclasses inherit the features of Window, such as a visible region on the screen. ScrollingWindow adds a scroll bar and an offset. The ability to factor out common features of several classes into a superclass can greatly reduce repetition within designs and programs and is one of the main advantages of OO technology.

Polymorphism means that the same operation may behave differently for different classes. The move operation, for example, behaves differently for a pawn than for the queen in a chess game. An operation is a procedure or transformation that an object performs or is subject to. RightJustify, display, and move are examples of operations. An implementation of an operation by a specific class is called a method. Because an OO operator is polymorphic, it may have more than one method implementing it, each for a different class of object.

In the real world, an operation is simply an abstraction of analogous behavior across different kinds of objects. Each object “knows how” to perform its own operations. In an OO programming language, however, the language automatically selects the correct method to implement an operation based on the name of the operation and the class of the object being operated on. The user of an operation need not be aware of how many methods exist to implement a given polymorphic operation. Developers can add new classes without changing existing code, as long as they provide methods for each applicable operation.

1.2 What Is OO Development?

This book is about OO development as a way of thinking about software based on abstractions that exist in the real world as well as in the program. In this context development refers to the software life cycle: analysis, design, and implementation. The essence of OO development is the identification and organization of application concepts, rather than their final representation in a programming language. Brooks observes that the hard part of software development is the manipulation of its essence, owing to the inherent complexity of the problem, rather than the accidents of its mapping into a particular language (Brooks-95).

This book does not explicitly address integration, maintenance, and enhancement, but a clean design in a precise notation facilitates the entire software life cycle. The OO concepts and notation used to express a design also provide useful documentation.
1.2.1 Modeling Concepts, Not Implementation

In the past, much of the OO community focused on programming languages, with the literature emphasizing implementation rather than analysis and design. OO programming languages were first useful in alleviating the inflexibility of traditional programming languages. In a sense, however, this emphasis was a step backward for software engineering—it focuses excessively on implementation mechanisms, rather than the underlying thought process that they support.

The real payoff comes from addressing front-end conceptual issues, rather than back-end implementation details. Design flaws that surface during implementation are more costly to fix than those that are found earlier. A premature focus on implementation restricts design choices and often leads to an inferior product. An OO development approach encourages software developers to work and think in terms of the application throughout the software life cycle. It is only when the inherent concepts of the application are identified, organized, and understood that the details of data structures and functions can be addressed effectively.

OO development is a conceptual process independent of a programming language until the final stages. OO development is fundamentally a way of thinking and not a programming technique. Its greatest benefits come from helping specifiers, developers, and customers express abstract concepts clearly and communicate them to each other. It can serve as a medium for specification, analysis, documentation, and interfacing, as well as for programming.

1.2.2 OO Methodology

We present a process for OO development and a graphical notation for representing OO concepts. The process consists of building a model of an application and then adding details to it during design. The same seamless notation is used from analysis to design to implementation, so that information added in one stage of development need not be lost or translated for the next stage. The methodology has the following stages.

- **System conception.** Software development begins with business analysts or users conceiving an application and formulating tentative requirements.
- **Analysis.** The analyst scrutinizes and rigorously restates the requirements from system conception by constructing models. The analyst must work with the requestor to understand the problem, because problem statements are rarely complete or correct. The analysis model is a concise, precise abstraction of what the desired system must do, not how it will be done. The analysis model should not contain implementation decisions. For example, a *Window* class in a workstation windowing system would be described in terms of its visible attributes and operations.

  The analysis model has two parts: the *domain model*, a description of the real-world objects reflected within the system; and the *application model*, a description of the parts of the application system itself that are visible to the user. For example, domain objects for a stockbroker application might include stock, bond, trade, and commission. Application objects might control the execution of trades and present the results. Application experts who are not programmers can understand and criticize a good model.

### 1.2. What Is OO Development?

- **System design.** The development team devise a high-level strategy—the *system architecture*—for solving the application problem. They also establish policies that will serve as a default for the subsequent, more detailed portions of design. The system designer must decide what performance characteristics to optimize, choose a strategy of attacking the problem, and make tentative resource allocations. For example, the system designer might decide that changes to the workstation screen must be fast and smooth, even when windows are moved or erased, and choose an appropriate communications protocol and memory buffering strategy.

- **Class design.** The class designer adds details to the analysis model in accordance with the system design strategy. The class designer elaborates both domain and application objects using the same OO concepts and notation, although they exist on different conceptual planes. The focus of class design is the data structures and algorithms needed to implement each class. For example, the class designer now determines data structures and algorithms for each of the operations of the *Window* class.

- **Implementation.** Implementers translate the classes and relationships developed during class design into a particular programming language, database, or hardware. Programming should be straightforward, because all of the hard decisions should have already been made. During implementation, it is important to follow good software engineering practice so that traceability to the design is apparent and so that the system remains flexible and extensible. For example, implementers would code the *Window* class in a programming language, using calls to the underlying graphics system on the workstation.

OO concepts apply throughout the system development life cycle, from analysis through design to implementation. You can carry the same classes from stage to stage without a change of notation, although they gain additional details in the later stages. The analysis and implementation models of *Window* are both correct, but they serve different purposes and represent a different level of abstraction. The same OO concepts of identity, classification, polymorphism, and inheritance apply throughout development.

Note that we are not suggesting a waterfall development process—first capturing requirements, then analyzing, then designing, and finally implementing. For any particular part of a system, developers must perform each stage in order, but they need not develop each part of the system in tandem. We advocate an iterative process—developing part of the system through several stages and then adding capability.

Some classes are not part of analysis but are introduced during design or implementation. For example, data structures such as *trees*, *hash tables*, and *linked lists* are rarely present in the real world and are not visible to users. Designers introduce them to support particular algorithms. Such data structure objects exist within a computer and are not directly observable.

We do not consider testing as a distinct step. Testing is important, but it must be part of an overall philosophy of quality control that occurs throughout the life cycle. Developers must check analysis models against reality. They must verify design models against various kinds of errors, in addition to testing implementations for correctness. Confining quality control to a separate step is more expensive and less effective.
1.2.3 Three Models

We use three kinds of models to describe a system from different viewpoints: the class model for the objects in the system and their relationships; the state model for the life history of objects; and the interaction model for the interactions among objects. Each model applies during all stages of development and acquires detail as development progresses. A complete description of a system requires models from all three viewpoints.

The **class model** describes the static structure of the objects in a system and their relationships. The class model defines the context for software development—the universe of discourse. The class model contains class diagrams. A **class diagram** is a graph whose nodes are classes and whose arcs are relationships among classes.

The **state model** describes the aspects of an object that change over time. The state model specifies and implements control with state diagrams. A **state diagram** is a graph whose nodes are states and whose arcs are transitions between states caused by events.

The **interaction model** describes how the objects in a system cooperate to achieve broader results. The interaction model starts with use cases that are then elaborated with sequence and activity diagrams. A **use case** focuses on the functionality of a system—that is, what a system does for users. A **sequence diagram** shows the objects that interact and the time sequence of their interactions. An **activity diagram** elaborates important processing steps.

The three models are separate parts of the description of a complete system but are cross-linked. The class model is most fundamental, because it is necessary to describe what is changing or transforming before describing when or how it changes.

1.3 OO Themes

Several themes pervade OO technology. Although these themes are not unique to OO systems, they are particularly well supported.

1.3.1 Abstraction

**Abstraction** lets you focus on essential aspects of an application while ignoring details. This means focusing on what an object is and does, before deciding how to implement it. Use of abstraction preserves the freedom to make decisions as long as possible by avoiding premature commitments to details. Most modern languages provide data abstraction, but inheritance and polymorphism add power. The ability to abstract is probably the most important skill required for OO development.

1.3.2 Encapsulation

**Encapsulation** (also **information hiding**) separates the external aspects of an object, that are accessible to other objects, from the internal implementation details, that are hidden from other objects. Encapsulation prevents portions of a program from becoming so interdependent that a small change has massive ripple effects. You can change an object's implementation without affecting the applications that use it. You may want to change the implementation of an object to improve performance, fix a bug, consolidate code, or support porting. Encapsulation is not unique to OO languages, but the ability to combine data structure and behavior in a single entity makes encapsulation cleaner and more powerful than in prior languages, such as Fortran, Cobol, and C.

1.3.3 Combining Data and Behavior

The caller of an operation need not consider how many implementations exist. Operator polymorphism shifts the burden of deciding what implementation to use from the calling code to the class hierarchy. For example, non-OO code to display the contents of a window must distinguish the type of each figure, such as polygon, circle, or text, and call the appropriate procedure to display it. An OO program would simply invoke the **draw** operation on each figure; each object implicitly decides which procedure to use, based on its class. Maintenance is easier, because the calling code need not be modified when a new class is added.

In an OO system, the data structure hierarchy matches the operation inheritance hierarchy (Figure 1.3).

![Figure 1.3 OO vs. prior approach. An OO approach has one unified hierarchy for both data and behavior.](image)

1.3.4 Sharing

OO techniques promote sharing at different levels. Inheritance of both data structure and behavior lets subclasses share common code. This sharing via inheritance is one of the main advantages of OO languages. More important than the savings in code is the conceptual clarity from recognizing that different operations are all really the same thing. This reduces the number of distinct cases that you must understand and analyze.

OO development not only lets you share information within an application, but also offers the prospect of reusing designs and code on future projects. OO development provides the tools, such as abstraction, encapsulation, and inheritance, to build libraries of reusable
components. Unfortunately, reuse has been overemphasized as a justification for OO technology. Reuse does not just happen; developers must plan by thinking beyond the immediate application and investing extra effort in a more general design.

1.3.5 Emphasis on the Essence of an Object

OO technology stresses what an object is, rather than how it is used. The uses of an object depend on the details of the application and often change during development. As requirements evolve, the features supplied by an object are much more stable than the ways it is used, hence software systems built on object structure are more stable in the long run. OO development places a greater emphasis on data structure and a lesser emphasis on procedure structure than functional-decomposition methodologies. In this respect, OO development is similar to information modeling techniques used in database design, although OO development adds the concept of class-dependent behavior.

1.3.6 Synergy

Identity, classification, polymorphism, and inheritance characterize OO languages. Each of these concepts can be used in isolation, but together they complement each other synergistically. The benefits of an OO approach are greater than they might seem at first. The emphasis on the essential properties of an object forces the developer to think more carefully and deeply about what an object is and does. The resulting system tends to be cleaner, more general, and more robust than it would be if the emphasis were only on the use of data and operations.

1.4 Evidence for Usefulness of OO Development

Our work on OO development began with internal applications at the General Electric Research and Development Center. We used OO techniques for developing compilers, graphics, user interfaces, databases, an OO language, CAD systems, simulations, metamodels, control systems, and other applications. We used OO models to document programs that are ill-structured and difficult to understand. Our implementation targets ranged from OO languages to non-OO languages to databases. We successfully taught this approach to others and used it to communicate with application experts.

Since the mid 1990s we have expanded our practice of OO technology beyond General Electric to companies throughout the world. When we wrote the first edition of this book, object orientation and OO modeling were relatively new approaches without much large-scale experience. OO technology can no longer be considered a fad or a speculative approach. It is now part of the computer science and software engineering mainstream.

The annual OOPSLA (Object-Oriented Programming Systems, Languages, and Applications), ECOOP (European Conference on Object-Oriented Programming), and TOOLS (Technology of Object-Oriented Languages and Systems) conferences are important forums for disseminating new OO ideas and application results. The conference proceedings describe many applications that have benefited from an OO approach. Articles on OO systems have also appeared in major publications, such as IEEE Computer and Communications of the ACM.

1.5 OO Modeling History

Our work at GE R&D led to the development of the Object Modeling Technique (OMT), which the previous edition of this book introduced in 1991. OMT was a success, but so were several other approaches. The popularity of OO modeling led to a new problem—a plethora of alternative notations. The notations expressed similar ideas but had different symbols, confusing developers and making communication difficult.

As a result, the software community began to focus on consolidating the various notations. In 1994 Jim Rumbaugh joined Rational (now part of IBM) and began working with Grady Booch on unifying the OMT and Booch notations. In 1995, Ivar Jacobson also joined Rational and added Objectory to the unification work.

In 1996 the Object Management Group (OMG) issued a request for proposals for a standard OO modeling notation. Several companies responded, and eventually the competing proposals were coalesced into a final proposal. Rational led the final proposal team, with Booch, Rumbaugh, and Jacobson deeply involved. The OMG unanimously accepted the resulting Unified Modeling Language (UML) as a standard in November 1997. The participating companies transferred UML rights to the OMG, which owns the trademark and specification for UML and controls its future development.

The UML was highly successful and replaced the other notations in most publications. Most of the authors of other methods adopted UML notation, willingly or because of market pressure. The UML has ended the OO notation wars and is now clearly the accepted OO notation. We have used UML in this book because it is now the standard notation.

In 2001 OMG members started work on a revision to add features missing from the initial specification and to fix problems that were discovered by experience with UML 1. This book is based on the UML 2.0 revision approved in 2004. For access to the official specification documents, see the OMG Web site at www.omg.org.

1.6 Organization of This Book

The remainder of this book is organized into four parts: modeling concepts, analysis/design, implementation, and software engineering. Appendices provide a glossary of terms and answer some of the exercises. The inside covers summarize the notation used in the book.

Part 1 explains OO concepts and presents a graphical notation for expressing them. Chapter 2 introduces modeling and three kinds of models—class, state, and interaction. Chapters 3 and 4 describe the class model, which deals with the structural "data" aspects of a system—these chapters are the heart of Part 1, and mastery of the class model is essential for successful OO development. Chapters 5 and 6 present the state model, which concerns the control aspects of a system. Chapters 7 and 8 describe the interaction model, which captures the interactions among different objects in a system. Chapter 9 summarizes the three models and how they relate to each other. The concepts dealt with in Part 1 permeate the software development cycle, applying equally to analysis, design, and implementation. The entire book uses the notation described in Part 1.
Part 2 shows how to prepare an OO model and use it to analyze and design a system. Chapter 10 summarizes the process, and then Chapter 11 discusses system conception, the invention of an application. Chapters 12 and 13 discuss analysis, the process of describing and understanding the application. Analysis begins with a problem statement from the customer. The analyst incorporates customer information and application knowledge to construct domain and application models. Chapter 14 addresses system design, which is primarily a task of partitioning a system into subsystems and making high-level policy decisions. Chapter 15 presents class design, the augmentation of the analysis model with design decisions. These decisions include the specification of algorithms, assigning functionality to objects, and optimization. Chapter 16 summarizes the process.

Part 3 addresses implementation, with Chapter 17 discussing issues apart from the target language. Chapters 18 and 19 address C++, Java, and databases. Chapter 20 presents guidelines for enhancing readability, reusability, and maintainability using good OO programming style.

Part 4 focuses on software engineering. Although Part 2 presents the stages in a linear order, as a book must, we do not believe that development should proceed in a waterfall fashion. Chapter 21 describes iterative development, in which the process stages are repeated multiple times to build the complete system. Chapter 22 provides advice for managing models. It is easiest to understand and to apply OO development on a new system, but most projects do not have the luxury of working on a clean slate. Chapter 23 describes issues involved in working with existing systems.

Most chapters contain exercises. Selected answers are included in the back of the book. We suggest that you try to work the exercises as you read this book, even if you are not a student. The exercises bring out many subtle points. They provide practice with OO technology and serve as a stepping stone to applications.

| abstraction | encapsulation | object-oriented (OO) |
| analysis | identity | polymorphism |
| class design | implementation | state model |
| class model | inheritance | system design |
| classification | interaction model |

Figure 1.4 Key concepts for Chapter 1

Bibliographic Notes
[Taylor-98] provides a well-written overview of OO technology. [Meyer-97] is also an informative source, even though it is primarily an OO language book. [Love-93] presents examples of industrial projects that have used OO technology.

The purpose of this book is to teach OO concepts and thinking, not serve as a UML reference manual (see [Rumbaugh-05] for that). A textbook should emphasize important concepts, not fine details. Therefore present the most useful aspects of the UML, but we do not try to describe everything. You will learn faster by focusing on core concepts.

We have made a similar condensation of the development process. The process we describe is simple and aimed at small and medium projects. It contains highlights of the Unified Process (see [Jacobson-99]).

The UML contains the concept of a classifier, a more general form of a class that abstracts various kinds of modeling entities. For most purposes, there is little difference between a class and a classifier. In this book, we use the word class in preference to classifier, because modelers will work mostly with classes.

References


Exercises
The number in parentheses next to each exercise indicates the difficulty, from 1 (easy) to 10 (very difficult).

1.1 (3) What major problems have you encountered during past software projects? Estimate what percentage of your time you spend on analysis, design, coding, and testing/debugging/fixing. How do you go about estimating how much effort a project will require?

1.2 (3) Recall a past system that you created. Briefly describe it. What obstacles did you encounter in the design? What software engineering methodology, if any, did you use? What were your reasons for choosing or not choosing a methodology? Are you satisfied with the system as it exists? How difficult is it to add new features to the system? Is it maintainable?

1.3 (3) Describe a recent large software system that was behind schedule, over budget, or failed to perform as expected. What factors were blamed? How could the failure have been avoided?

1.4 (3) From a user's point of view, criticize a hardware or software system that has a flaw that especially annoys you. For example, some cars require the bumper to be removed to replace a tail light. Describe the system, the flaw, how it was overlooked, and how it could have been avoided with a bit more thought during design.
1.5 (5) All objects have identity and are distinguishable. However, for large collections of objects, it may not be a trivial matter to devise a scheme to distinguish them. Furthermore, a scheme may depend on the purpose of the distinction. For each of the following collections of objects, describe how they could be distinguished.

a. All persons in the world for the purpose of sending mail
b. All persons in the world for the purpose of criminal investigations
c. All customers with safe deposit boxes in a given bank
d. All telephones in the world for making telephone calls
e. All customers of a telephone company for billing purposes
f. All electronic mail addresses throughout the world
g. All employees of a company to restrict access for security reasons

1.6 (4) Prepare a list of classes that you would expect each of the following systems to handle.

a. A program for laying out a newspaper
b. A program to compute and store bowling scores
c. A telephone voice mail system with delivery options, message forwarding, and group lists
d. A controller for a video cassette recorder
e. A catalog store order entry system

1.7 (6) Classes and operations are listed below. For each class, select the operations that make sense for objects in that class. You may place an operation in multiple classes. Discuss the behavior of each operation.

Classes:
- variable-length array — ordered collection of objects, indexed by an integer, whose size can vary at run time
- symbol table — a table that maps text keywords into descriptors
- set — unordered collection of objects with no duplicates

Operations:
- append — add an object to the end of a collection
- copy — make a copy of a collection
- count — return the number of elements in a collection
- delete — remove an element from a collection
- index — retrieve an object from a collection at a given position
- intersect — determine the common elements of two collections
- insert — place an object into a collection at a given position
- update — add an element to a collection, writing over whatever is already there

1.8 (4) Discuss what the classes in each of the following lists have in common. You may add more classes to each list.

a. scanning electron microscope, eyeglasses, telescope, bomb sight, binoculars
b. pipe, check valve, faucet, filter, pressure gauge
c. bicycle, sailboat, car, truck, airplane, glider, motorcycle, horse
d. nail, screw, bolt, rivet
e. tent, cave, shed, garage, barn, house, skyscraper

Part 1

Modeling Concepts

Chapter 2  Modeling as a Design Technique  15
Chapter 3  Class Modeling  21
Chapter 4  Advanced Class Modeling  60
Chapter 5  State Modeling  90
Chapter 6  Advanced State Modeling  110
Chapter 7  Interaction Modeling  131
Chapter 8  Advanced Interaction Modeling  147
Chapter 9  Concepts Summary  161

Part 1 describes the concepts and notations involved in object-oriented modeling. The concepts and notation apply to analysis, design, and implementation.

Chapter 2 discusses modeling in general and then introduces the three kinds of object-oriented models—class, state, and interaction.

Chapter 3 presents the class model which describes the static structure of a system. The class model provides the context for the other two kinds of models. Chapter 4 covers advanced class modeling concepts that you can skip upon a first reading of the book.

Chapter 5 explains the state model which describes the aspects of a system that change over time as well as control behavior. Chapter 6 covers advanced state modeling concepts that you can also skip upon a first reading.

Chapter 7 presents the interaction model and completes the treatment of the three models. The interaction model describes how objects collaborate to achieve overall results. Chapter 8 is an advanced chapter on interaction modeling that you can skip upon an initial reading.

Chapter 9 briefly summarizes the three kinds of models and how they relate to each other.

After reading Part 1, you will understand object-oriented concepts and the UML notation for expressing them. You will be ready to apply the concepts to software development in subsequent parts of the book.
Modeling as a Design Technique

A model is an abstraction of something for the purpose of understanding it before building it. Because a model omits nonessential details, it is easier to manipulate than the original entity. Abstraction is a fundamental human capability that permits us to deal with complexity. Engineers, artists, and craftsmen have built models for thousands of years to try out designs before executing them. Development of hardware and software systems is no exception. To build complex systems, the developer must abstract different views of the system, build models using precise notations, verify that the models satisfy the requirements of the system, and gradually add detail to transform the models into an implementation.

2.1 Modeling

Designers build many kinds of models for various purposes before constructing things. Examples include architectural models to show customers, airplane scale models for wind-tunnel tests, pencil sketches for composition of oil paintings, blueprints of machine parts, storyboards of advertisements, and outlines of books. Models serve several purposes.

- **Testing a physical entity before building it.** The medieval masons did not know modern physics, but they built scale models of the Gothic cathedrals to test the forces on the structure. Engineers test scale models of airplanes, cars, and boats in wind tunnels and water tanks to improve their dynamics. Recent advances in computation permit the simulation of many physical structures without the need to build physical models. Not only is simulation cheaper, but it provides information that is too fleeting or inaccessible to be measured from a physical model. Both physical models and computer models are usually cheaper than building a complete system and enable early correction of flaws.

- **Communication with customers.** Architects and product designers build models to show their customers. Mock-ups are demonstration products that imitate some or all of the external behavior of a system.
2.3 The Three Models

The three kinds of models separate a system into distinct views. The different models are not completely independent—a system is more than a collection of independent parts—but each model can be examined and understood by itself to a large extent. The different models have limited and explicit interconnections. Of course, it is always possible to create bad designs in which the three models are so intertwined that they cannot be separated, but a good design isolates the different aspects of a system and limits the coupling between them.

Each of the three models evolves during development. First analysts construct a model of the application without regard for eventual implementation. Then designers add solution constructs to the model. Implementers code both application and solution constructs. The word *model* has two dimensions—a view of a system (class model, state model, or interaction model) and a stage of development (analysis, design, or implementation). The meaning is generally clear from context.

2.3.1 Class Model

The *class model* describes the structure of objects in a system—their identity, their relationships to other objects, their attributes, and their operations. The class model provides context for the state and interaction models. Changes and interactions are meaningless unless there is something to be changed or with which to interact. Objects are the units into which we divide the world, the molecules of our models.

Our goal in constructing a class model is to capture those concepts from the real world that are important to an application. In modeling an engineering problem, the class model should contain terms familiar to engineers; in modeling a business problem, terms from the business; in modeling a user interface, terms from the application. An analysis model should not contain computer constructs unless the application being modeled is inherently a computer problem, such as a compiler or an operating system. The design model describes how to solve a problem and may contain computer constructs.

Class diagrams express the class model. Generalization lets classes share structure and behavior, and associations relate the classes. Classes define the attribute values carried by each object and the operations that each object performs or undergoes.

2.3.2 State Model

The *state model* describes those aspects of objects concerned with time and the sequencing of operations—events that mark changes, states that define the context for events, and the organization of events and states. The state model captures control, the aspect of a system that describes the sequences of operations that occur, without regard for what the operations do, what they operate on, or how they are implemented.

State diagrams express the state model. Each state diagram shows the state and event sequences permitted in a system for one class of objects. State diagrams refer to the other models. Actions and events in a state diagram become operations on objects in the class model. References between state diagrams become interactions in the interaction model.
2.3.3 Interaction Model

The interaction model describes interactions between objects—how individual objects collaborate to achieve the behavior of the system as a whole. The state and interaction models describe different aspects of behavior, and you need both to describe behavior fully.

Use cases, sequence diagrams, and activity diagrams document the interaction model. Use cases document major themes for interaction between the system and outside actors. Sequence diagrams show the objects that interact and the time sequence of their interactions. Activity diagrams show the flow of control among the processing steps of a computation.

2.3.4 Relationship Among the Models

Each model describes one aspect of the system but contains references to the other models. The class model describes data structure on which the state and interaction models operate. The operations in the class model correspond to events and actions. The state model describes the control structure of objects. It shows decisions that depend on object values and causes actions that change object values and state. The interaction model focuses on the exchanges between objects and provides a holistic overview of the operation of a system.

There are occasional ambiguities about which model should contain a piece of information. This is natural, because any abstraction is only a rough cut at reality; something will inevitably straddle the boundaries. Some properties of a system may be poorly represented by the models. This is also normal, because no abstraction is perfect; the goal is to simplify the system description without loading down the model with so many constructs that it becomes a burden and not a help. For those things that the model does not adequately capture, natural language or application-specific notation is still perfectly acceptable.

2.4 Chapter Summary

Models are abstractions built to understand a problem before implementing a solution. All abstractions are subsets of reality selected for a particular purpose.

We recommend three kinds of models. The class model describes the static structure of a system in terms of classes and relationships. The state model describes the control structure of a system in terms of events and states. The interaction model describes how individual objects collaborate to achieve the behavior of the system as a whole. Different problems place different emphasis on the three kinds of models.

- Abstraction modeling
- Class model relationship among models
- Interaction model state model

Figure 2.1 Key concepts for Chapter 2

Bibliographic Notes

The first edition of this book also had three models (object, dynamic, and functional), but they were organized differently than those in this second edition.

The object model in the first edition is the same as the class model presented here. We have changed the name to class model to stress that the modeling entities are descriptors (classes and relationships) rather than instances (objects and links). Our presentation of the class model in this book also includes constraint modeling, which was missing from the first edition.

Similarly, the dynamic model in the first edition is the same as the state model in this book. We changed the name to state model to avoid confusion with other representations of dynamic behavior. The UML contains multiple kinds of models with various degrees of overlap—we cover the most important ones in this book.

We have dropped the functional model from the second edition. Certainly, the eventual software has functionality, but we seldom capture it with data flow diagrams as was shown in the first edition. We included data flow diagrams in the first edition for continuity with the structured analysis/structured design approach of the past. The functional model was not as useful as we envisioned, so we have now dropped it.

In its place, the second edition adds the interaction model. State diagrams do express dynamic behavior fully, but often in an inscrutable manner. Each state diagram focuses on a single class. When many classes have a significant state diagram, it can be difficult to understand an entire system. The interaction model focuses on collaboration and helps a software developer obtain a more comprehensive understanding than with state diagrams alone.

Exercises

2.1 (1) Some characteristics of an automotive tire are its size, material, internal construction (bias, steel belted, for example), tread design, cost, expected life, and weight. Which factors are important in deciding whether or not to buy a tire for your car? Which ones might be relevant to someone simulating the performance of a computerized anti-skid system for cars? Which ones are important to someone constructing a swing for a child?

2.2 (2) Suppose your bathroom sink is clogged and you have decided to try to unplug it by pushing a wire into the drain. You have several types of wire available around the house, some insulated and some not. Which of the following wire characteristics would you need to consider in selecting a wire for the job? Explain your answers.
   a. Immunity to electrical noise
   b. Color of the insulation
   c. Resistance of the insulation to salt water
   d. Resistance of the insulation to fire
   e. Cost
   f. Stiffness
   g. Ease of stripping the insulation
2.3 (3) Wire is used in the following applications. For each application, prepare a list of wire characteristics that are relevant and explain why each is important for the application.

a. Selecting wire for a transatlantic cable
b. Choosing wire that you will use to create colorful artwork
c. Designing the electrical system for an airplane
d. Hanging a bird feeder from a tree
e. Designing a piano
f. Designing the filament for a light bulb

2.4 (3) If you were designing a protocol for transferring computer files from one computer to another over telephone lines, which of the following details would you select as relevant? Explain how they are relevant.

a. Electrical noise on the communication lines
b. The speed at which serial data is transmitted
c. Availability of a database
d. Availability of a good full-screen editor
e. Buffering and flow control, such as an XON/XOFF protocol to regulate an incoming stream of data
f. Number of tracks and sectors on a disk drive
g. Character interpretation, such as special handling of control characters
h. File organization, linear stream of bytes versus record-oriented, for example
i. Math co-processor

2.5 (2) There are several models used in the analysis and design of electrical motors. An electrical model involves voltages, currents, electromagnetic fields, inductance, and resistance. A mechanical model considers stiffness, density, motion, forces, and torques. A thermal model handles heat dissipation and heat transfer. A fluid model describes the flow of cooling air. Which model(s) can answer the following questions? Discuss your conclusions.

a. How much power is required to run a motor? How much of it is wasted as heat?
b. How much does a motor weigh?
c. How hot does a motor get?
d. How much vibration does a motor create?
e. How long will it take for the bearings of a motor to wear out?

2.6 (3) Decide which model(s) (class, state, interaction) are relevant for the following aspects of a computer chess player. A video display will show the board and pieces. A cursor controlled by a mouse will indicate human moves. Of course, in some cases, more than one model may apply. Explain your answers.

a. User interface that displays computer moves and accepts human moves
b. Representation of a configuration of pieces on the board
c. Consideration of a sequence of possible legal moves
d. Validation of a move requested by the human player

3 Class Modeling

A class model captures the static structure of a system by characterizing the objects in the system, the relationships between the objects, and the attributes and operations for each class of objects. The class model is the most important of the three models. We emphasize building a system around objects rather than around functionality; because an object-oriented system more closely corresponds to the real world and is consequently more resilient with respect to change. Class models provide an intuitive graphic representation of a system and are valuable for communicating with customers.

Chapter 3 discusses basic class modeling concepts that will be used throughout the book. We define each concept, present the corresponding UML notation, and provide examples. Some important concepts that we consider are object, class, link, association, generalization, and inheritance. You should master the material in this chapter before proceeding in the book.

3.1 Object and Class Concepts

3.1.1 Objects

The purpose of class modeling is to describe objects. For example, Joe Smith, Simplex company, process number 7648, and the top window are objects.

An object is a concept, abstraction, or thing with identity that has meaning for an application. Objects often appear as proper nouns or specific references in problem descriptions and discussions with users. Some objects have real-world counterparts (Albert Einstein and the General Electric company), while others are conceptual entities (simulation run 1234 and the formula for solving a quadratic equation). Still others (binary tree 634 and the array bound to variable a) are introduced for implementation reasons and have no correspondence to physical reality. The choice of objects depends on judgment and the nature of a problem; there can be many correct representations.
All objects have identity and are distinguishable. Two apples with the same color, shape, and texture are still individual apples; a person can eat one and then eat the other. Similarly, identical twins are two distinct persons, even though they may look the same. The term identity means that objects are distinguished by their inherent existence and not by descriptive properties that they may have.

3.1.2 Classes
An object is an instance—or occurrence—of a class. A class describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics. Person, company, process, and window are all classes. Each person has name and birthdate and may work at a job. Each process has an owner, priority, and list of required resources. Classes often appear as common nouns and noun phrases in problem descriptions and discussions with users.

Objects in a class have the same attributes and forms of behavior. Most objects derive their individuality from differences in their attribute values and specific relationships to other objects. However, objects with identical attribute values and relationships are possible. The choice of classes depends on the nature and scope of an application and is a matter of judgment.

The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior. For example, a barn and a horse may both have a cost and an age. If barn and horse were regarded as purely financial assets, they could belong to the same class. If the developer took into consideration that a person paints a barn and feeds a horse, they would be modeled as distinct classes. The interpretation of semantics depends on the purpose of each application and is a matter of judgment.

Each object “knows” its class. Most OO programming languages can determine an object’s class at run time. An object’s class is an implicit property of the object.

If objects are the focus of modeling, why bother with classes? The notion of abstraction is at the heart of the matter. By grouping objects into classes, we abstract a problem. Abstraction gives modeling its power and ability to generalize from a few specific cases to a host of similar cases. Common definitions (such as class name and attribute names) are stored once per class rather than once per instance. You can write operations once for each class, so that all the objects in the class benefit from code reuse. For example, all ellipses share the same procedures to draw them, compute their areas, and test for intersection with a line; polygons would have a separate set of procedures. Even special cases, such as circles and squares, can use the general procedures, though more efficient procedures are possible.

3.1.3 Class Diagrams
We began this chapter by discussing some basic modeling concepts, specifically object and class. We have described these concepts with examples and prose. This approach is vague and insufficient for dealing with the complexity of applications. We need a means for expressing models that is coherent, precise, and easy to formulate. There are two kinds of models of structure—class diagrams and object diagrams.

3.1.4 Values and Attributes
A value is a piece of data. You can find values by examining problem documentation for examples. An attribute is a named property of a class that describes a value held by each object of the class. You can find attributes by looking for adjectives or by abstracting typical values. The following analogy holds: Object is to class as value is to attribute. Structural constructs—that is, classes and relationships (to be explained)—dominate class models. Attributes are of lesser importance and serve to elaborate classes and relationships.

Name, birthdate, and weight are attributes of Person objects. Color, modelYear, and weight are attributes of Car objects. Each attribute has a value for each object. For example, attribute birthdate has value “21 October 1983” for object JoeSmith. Paraphrasing, Joe Smith was born on 21 October 1983. Different objects may have the same or different values for a given attribute. Each attribute name is unique within a class (as opposed to being unique across all classes). Thus class Person and class Car may each have an attribute called weight.
Chapter 3 / Class Modeling

Do not confuse values with objects. An attribute should describe values, not objects. Unlike objects, values lack identity. For example, all occurrences of the integer "17" are indistinguishable, as are all occurrences of the string "Canada." The country Canada is an object, whose name attribute has the value "Canada" (the string).

Figure 3.2 shows modeling notation. Class Person has attributes name and birthdate. Name is a string and birthdate is a date. One object in class Person has the value "Joe Smith" for name and the value "21 October 1983" for birthdate. Another object has the value "Mary Sharp" for name and the value "16 March 1950" for birthdate.

Do not confuse internal identifiers with real-world attributes. Internal identifiers are purely an implementation convenience and have no application meaning. In contrast, tax payer number, license plate number, and telephone number are not internal identifiers because they have meaning in the real world. Rather they are legitimate attributes.

3.1 Object and Class Concepts

3.1.5 Operations and Methods
An operation is a function or procedure that may be applied to or by objects in a class. Hire, fire, and payDividend are operations on class Company. Open, close, hide, and redisplay are operations on class Window. All objects in a class share the same operations.

Each operation has a target object as an implicit argument. The behavior of the operation depends on the class of its target. An object "knows" its class, and hence the right implementation of the operation.

The same operation may apply to many different classes. Such an operation is polymorphic; that is, the same operation takes on different forms in different classes. A method is the implementation of an operation for a class. For example, the class File may have an operation print. You could implement different methods to print ASCII files, print binary files, and print digitized picture files. All these methods logically perform the same task—printing a file; thus you may refer to them by the generic operation print. However, a different piece of code may implement each method.

An operation may have arguments in addition to its target object. Such arguments may be placeholders for values, or for other objects. The choice of a method depends entirely on the class of the target object and not on any object arguments that an operation may have. (A few OO languages, notably CLOS, permit the choice of method to depend on any number of arguments, but such generality leads to considerable semantic complexity, which we shall not explore.)

When an operation has methods on several classes, it is important that the methods all have the same signature—the number and types of arguments and the type of result value. For example, print should not have fileName as an argument for one method and filePointer for another. The behavior of all methods for an operation should have a consistent intent. It is best to avoid using the same name for two operations that are semantically different, even if they apply to distinct sets of classes. For example, it would be unwise to use the name invert to describe both a matrix inversion and turning a geometric figure upside-down. In a large project, some form of name scoping may be necessary to accommodate accidental name clashes, but it is best to avoid any possibility of confusion.

In Figure 3.4, the class Person has attributes name and birthdate and operations changeJob and changeAddress. Name, birthdate, changeJob, and changeAddress are features of Person. Feature is a generic word for either an attribute or operation. Similarly, File has a print operation. GeometricObject has move, select, and rotate operations. Move has argument delta, which is a Vector; select has one argument p, which is of type Point and returns a Boolean; and rotate has argument angle, which is an input of type float with a default value of 0.0.

The UML notation is to list operations in the third compartment of the class box. Our convention is to list the operation name in regular face, left align the name in the box, and use a lowercase letter for the first letter. Optional details, such as an argument list and result type, may follow each operation name. Parentheses enclose an argument list; commas separate the arguments. A colon precedes the result type. An empty argument list in parentheses shows explicitly that there are no arguments; otherwise you cannot draw conclusions. We do not list operations for objects, because they do not vary among objects of the same class.
3.2 Link and Association Concepts

Links and associations are the means for establishing relationships among objects and classes.

3.2.1 Links and Associations

A link is a physical or conceptual connection among objects. For example, Joe Smith WorksFor Simplex company. Most links relate two objects, but some links relate three or more objects. This chapter discusses only binary associations; Chapter 4 discusses n-ary associations. Mathematically, we define a link as a tuple—that is, a list of objects. A link is an instance of an association.

An association is a description of a group of links with common structure and common semantics. For example, a person WorksFor a company. The links of an association connect objects from the same classes. An association describes a set of potential links in the same way that a class describes a set of potential objects. Links and associations often appear as verbs in problem statements.

Figure 3.7 is an excerpt of a model for a financial application. Stock brokerage firms need to perform tasks such as recording ownership of various stocks, tracking dividends, alerting customers to changes in the market, and computing margin requirements. The top portion of the figure shows a class diagram and the bottom shows an object diagram.

Figure 3.7 Many-to-many association. An association describes a set of potential links in the same way that a class describes a set of potential objects.
In the class diagram, a person may own stock in zero or more companies; a company may have multiple persons owning its stock. The object diagram shows some examples. John, Mary, and Sue own stock in the GE company. Sue and Alice own stock in the IBM company. Jeff does not own stock in any company and thus has no link. The asterisk is a multiplicity symbol. Multiplicity specifies the number of instances of one class that may relate to a single instance of another class and is discussed in the next section.

The UML notation for a link is a line between objects; a line may consist of several line segments. If the link has a name, it is underlined. For example, John owns stock in the GE company. An association connects related classes and is also denoted by a line (with possibly multiple line segments). For example, persons own stock in companies. Our convention is to show link and association names in italics and to confine line segments to a rectilinear grid. It is good to arrange the classes in an association to read from left-to-right, if possible.

The association name is optional, if the model is unambiguous. Ambiguity arises when a model has multiple associations among the same classes (person works for company and person owns stock in company). When there are multiple associations, you must use association names or association end names (Section 3.2.3) to resolve the ambiguity.

Associations are inherently bidirectional. The name of a binary association usually reads in a particular direction, but the binary association can be traversed in either direction. For example, WorksFor connects a person to a company. The inverse of WorksFor could be called Employs, and it connects a company to a person. In reality, both directions of traversal are equally meaningful and refer to the same underlying association; it is only the names that establish a direction.

Developers often implement associations in programming languages as references from one object to another. A reference is an attribute in one object that refers to another object. For example, a data structure for Person might contain an attribute employer that refers to a Company object, and a Company object might contain an attribute employees that refers to a set of Person objects. Implementing associations as references is perfectly acceptable, but you should not model associations this way.

A link is a relationship among objects. Modeling a link as a reference disguises the fact that the link is not part of either object by itself, but depends on both of them together. A company is not part of a person, and a person is not part of a company. Furthermore, using a pair of matched references, such as the reference from Person to Company and the reference from Company to a set of Persons, hides the fact that the forward and inverse references depend on each other. Therefore, you should model all connections among classes as associations, even in designs for programs.

The OO literature emphasizes encapsulation, that implementation details should be kept private to a class, and we certainly agree with this. Associations are important, precisely because they break encapsulation. Associations cannot be private to a class, because they transcend classes. Failure to treat associations on an equal footing with classes can lead to programs containing hidden assumptions and dependencies. Such programs are difficult to extend and the classes are difficult to reuse.

Although modeling treats associations as bidirectional, you do not have to implement them in both directions. You can readily implement associations as references if they are only traversed in a single direction. Chapter 17 discusses some trade-offs to consider when implementing associations.

### 3.2.2 Multiplicity

Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. Multiplicity constraints the number of related objects. The literature often describes multiplicity as being "one" or "many," but more generally it is a (possibly infinite) subset of the nonnegative integers. UML diagrams explicitly list multiplicity at the ends of association lines. The UML specifies multiplicity with an interval, such as "1..*" (exactly one), "1.." (one or more), or "3..5" (three to five, inclusive). The special symbol "*" is a shorthand notation that denotes "many" (zero or more).

Figure 3.7 illustrates many-to-many multiplicity. A person may own stock in many companies. A company may have multiple persons holding its stock. In this particular case, John and Mary own stock in the GE company; Alice owns stock in the IBM company; Sue owns stock in both companies; Jeff does not own any stock. GE stock is owned by three persons; IBM stock is owned by two persons.

Figure 3.8 shows a one-to-one association and some corresponding links. Each country has one capital city. A capital city administers one country. (In fact, some countries, such as the Netherlands and Switzerland, have more than one capital city for different purposes. If this fact were important, the model could be modified by changing the multiplicity or by providing a separate association for each kind of capital city.)

![Class diagram](image)

![Object diagram](image)

**Figure 3.8** One-to-one association. Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

Figure 3.9 illustrates zero-or-one multiplicity. A workstation may have one of its windows designated as the console to receive general error messages. It is possible, however, that no console window exists. (The word "console" on the diagram is an association end name, discussed in Section 3.2.3.)
3.2 Link and Association Concepts

Figure 3.9 Zero-or-one multiplicity. It may be optional whether an object is involved in an association.

Do not confuse "multiplicity" with "cardinality." Multiplicity is a constraint on the size of a collection; cardinality is the count of elements that are actually in a collection. Therefore, multiplicity is a constraint on the cardinality.

A multiplicity of "many" specifies that an object may be associated with multiple objects. However, for each association there is at most one link between a given pair of objects (except for bags and sequences, see Section 3.2.5). As Figure 3.10 and Figure 3.11 show, if you want two links between the same objects, you must have two associations.

Multiplicity depends on assumptions and how you define the boundaries of a problem. Vague requirements often make multiplicity uncertain. Do not worry excessively about multiplicity early in software development. First determine classes and associations, then decide on multiplicity. If you omit multiplicity notation from a diagram, multiplicity is considered to be unspecified.

Multiplicity often exposes hidden assumptions built into a model. For example, is the WorksFor association between Person and Company one-to-many or many-to-many? It depends on the context. A tax collection application would permit a person to work for multiple companies. On the other hand, the member records for an auto workers' union may consider second jobs irrelevant. Class diagrams help to elicit these hidden assumptions, making them visible and subject to scrutiny.

The most important multiplicity distinction is between "one" and "many." Underestimating multiplicity can restrict the flexibility of an application. For example, many programs cannot accommodate persons with multiple phone numbers. On the other hand, overestimating multiplicity imposes overhead and requires the application to supply additional information to distinguish among the members of a "many" set. In a true hierarchical organization, for example, it is better to represent "boss" with a multiplicity of "zero or one," rather than allow for nonexistent matrix management.

3.2.3 Association End Names

Our discussion of multiplicity implicitly referred to the ends of associations. For example, a one-to-many association has two ends—an end with a multiplicity of "one" and an end with a multiplicity of "many." The notion of an association end is an important concept in the UML. You can not only assign a multiplicity to an association end, but you can give it a name as well. (Chapter 4 discusses additional properties of association ends.)

Association end names often appear as nouns in problem descriptions. As Figure 3.12 shows, a name appears next to the association end. In the figure Person and Company participate in association WorksFor. A person is an employee with respect to a company; a company is an employer with respect to a person. Use of association end names is optional, but it is often easier and less confusing to assign association end names instead of, or in addition to, association names.

Association end names are especially convenient for traversing associations, because you can treat each one as a pseudo attribute. Each end of a binary association refers to an object or set of objects associated with a source object. From the point of view of the source object, traversal of the association is an operation that yields related objects. Association end names provide a means of traversing an association, without explicitly mentioning the association. Section 3.5 talks further about traversing class models.

Association end names are necessary for associations between two objects of the same class. For example, in Figure 3.13 contains and contains distinguish the two usages of Directory in the self-association. A directory may contain many lesser directories and may optionally be contained itself. Association end names can also distinguish multiple associations between the same pair of classes. In Figure 3.13 each directory has exactly one user who is an owner and many users who are authorized to use the directory. When there is only a single association between a pair of distinct classes, the names of the classes often suffice, and you may omit association end names.
3.2 Link and Association Concepts

Figure 3.15 Ordering the objects for an association end. Ordering sometimes occurs for “many” multiplicity.

3.2.5 Bags and Sequences

Ordinarily a binary association has at most one link for a pair of objects. However, you can permit multiple links for a pair of objects by annotating an association end with \{bag\} or \{sequence\}. A bag is a collection of elements with duplicates allowed. A sequence is an ordered collection of elements with duplicates allowed. In Figure 3.16 an itinerary is a sequence of airports and the same airport can be visited more than once. Like the \{ordered\} indication, \{bag\} and \{sequence\} are permitted only for binary associations.

Figure 3.16 An example of a sequence. An itinerary may visit multiple airports, so you should use \{sequence\} and not \{ordered\}.

UML1 did not permit multiple links for a pair of objects. Some modelers misunderstood this restriction with ordered association ends and constructed incorrect models, assuming that there could be multiple links. With UML2 the modeler’s intent is now clear. If you specify \{bag\} or \{sequence\}, then there can be multiple links for a pair of objects. If you omit these annotations, then the association has at most one link for a pair of objects.

Note that the \{ordered\} and the \{sequence\} annotations are the same, except that the first disallows duplicates and the other allows them. A sequence association is an ordered bag, while an ordered association is an ordered set.

3.2.6 Association Classes

Just as you can describe the objects of a class with attributes, so too you can describe the links of an association with attributes. The UML represents such information with an association class. An association class is an association that is also a class. Like the links of an association, the instances of an association class derive identity from instances of the constituent classes. Like a class, an association class can have attributes and operations and participate in associations. You can find association classes by looking for adverbs in a problem statement or by abstracting known values.

In Figure 3.17, \texttt{accessPermission} is an attribute of \texttt{AccessibleBy}. The sample data at the bottom of the figure shows the value for each link. The UML notation for an association class is a box (a class box) attached to the association by a dashed line.

Figure 3.13 Association end names. Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

Association end names let you unify multiple references to the same class. When constructing class diagrams you should properly use association end names and not introduce a separate class for each reference, as Figure 3.14 shows. In the wrong model, two instances represent a person with a child, one for the child and one for the parent. In the correct model, one person instance participates in two or more links, twice as a parent and zero or more times as a child. (In the correct model, we must show a child as having an optional parent, so that the recursion eventually terminates.)

Because association end names distinguish objects, all names on the far end of associations attached to a class must be unique. Although the name appears next to the destination object on an association, it is really a pseudo attribute of the source class and must be unique within it. For the same reason, no association end name should be the same as an attribute name of the source class.

3.2.4 Ordering

Often the objects on a “many” association end have no explicit order, and you can regard them as a set. Sometimes, however, the objects have an explicit order. For example, Figure 3.15 shows a workstation screen containing a number of overlapping windows. Each window on a screen occurs at most once. The windows have an explicit order, so only the topmost window is visible at any point on the screen. The ordering is an inherent part of the association. You can indicate an ordered set of objects by writing \"{ordered}\" next to the appropriate association end.
Many-to-many associations provide a compelling rationale for association classes. Attributes for such associations unmistakably belong to the link and cannot be ascribed to either object. In Figure 3.17, accessPermission is a joint property of File and User and cannot be attached to either File or User alone without losing information.

Figure 3.18 presents attributes for two one-to-many associations. Each person working for a company receives a salary and has a job title. The boss evaluates the performance of each worker. Attributes may also occur for one-to-one associations.

Figure 3.19 shows how it is possible to fold attributes for one-to-one and one-to-many associations into the class opposite a "one" end. This is not possible for many-to-many associations. As a rule, you should not fold such attributes into a class because the multiplicity of the association might change. Either form in Figure 3.19 can express a one-to-many association. However, only the association class form remains correct if the multiplicity of WorksFor is changed to many-to-many.

Figure 3.20 shows an association class participating in an association. Users may be authorized on many workstations. Each authorization carries a priority and access privileges. A user has a home directory for each authorized workstation, but several workstations and users can share the same home directory. Association classes are an important aspect of class modeling because they let you specify identity and navigation paths precisely.
Chapter 3 / Class Modeling

Do not confuse an association class with an association that has been promoted to a class. Figure 3.21 highlights the difference. The association class has only one occurrence for each pairing of Person and Company. In contrast there can be any number of occurrences of a Purchase for each Person and Company. Each purchase is distinct and has its own quantity, date, and cost.

3.2.7 Qualified Associations
A qualified association is an association in which an attribute called the qualifier disambiguates the objects for a “many” association end. It is possible to define qualifiers for one-to-many and many-to-many associations. A qualifier selects among the target objects, reducing the effective multiplicity, from “many” to “one.” Qualified associations with a target multiplicity of “one” or “zero-or-one” specify a precise path for finding the target object from the source object.

Figure 3.22 illustrates the most common use of a qualifier — for associations with one-to-many multiplicity. A bank services multiple accounts. An account belongs to a single bank. Within the context of a bank, the account number specifies a unique account. Bank and Account are classes and accountNumber is the qualifier. Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one.

![Figure 3.22 Qualified association. Qualification increases the precision of a model.](image)

Both models are acceptable, but the qualified model adds information. The qualified model adds a multiplicity constraint, that the combination of a bank and an account number yields at most one account. The qualified model conveys the significance of account number in traversing the model, as methods will reflect. You first find the bank and then specify the account number to find the account.

The notation for a qualifier is a small box on the end of the association line near the source class. The qualifier box may grow out of any side (top, bottom, left, right) of the source class. The source class plus the qualifier yields the target class. In Figure 3.22 Bank + accountNumber yields an Account, therefore accountNumber is listed in a box contiguous to Bank.

Figure 3.23 provides another example of qualification. A stock exchange lists many companies. However, a stock exchange lists only one company with a given ticker symbol. A company may be listed on many stock exchanges, possibly under different symbols. (We are assuming this is true. If every stock had a single ticker symbol that was invariant across exchanges, we would make tickerSymbol an attribute of Company.)

3.3 Generalization and Inheritance

3.3.1 Definition
Generalization is the relationship between a class (the superclass) and one or more variations of the class (the subclasses). Generalization organizes classes by their similarities and differences, structuring the description of objects. The superclass holds common attributes, operations, and associations; the subclasses add specific attributes, operations, and associations. Each subclass is said to inherit the features of its superclass. Generalization is sometimes called the “is-a” relationship, because each instance of a subclass is an instance of the superclass as well.

Simple generalization organizes classes into a hierarchy; each subclass has a single immediate superclass. (Chapter 4 discusses a more complex form of generalization in which a subclass may have multiple immediate subclasses.) There can be multiple levels of generalizations.

Figure 3.24 shows several examples of generalization for equipment. Each piece of equipment is a pump, heat exchanger, or tank. There are several kinds of pumps: centrifugal, diaphragm, and plunger. There are several kinds of tanks: spherical, pressurized, and floating roof. The fact that the tank generalization symbol is drawn below the pump generalization symbol is not significant. Several objects are displayed at the bottom of the figure. Each object inherits features from one class at each level of the generalization. Thus P101 embodies the features of equipment, pump, and diaphragm pump. E302 has the properties of equipment and heat exchanger.

A large hollow arrowhead denotes generalization. The arrowhead points to the superclass. You may directly connect the superclass to each subclass, but we normally prefer to group subclasses as a tree. For convenience, you can rotate the triangle and place it on any side, but if possible you should draw the superclass on top and the subclasses on the bottom. The curly braces denote a UML comment, indicating that there are additional subclasses that the diagram does not show.
class. Each subclass not only inherits all the features of its ancestors but adds its own specific features as well. For example, Pump adds attributes suctionPressure, dischargePressure, and flowRate, which other kinds of equipment do not share.

Figure 3.25 shows classes of geometric figures. This example has more of a programming flavor and emphasizes inheritance of operations. Move, select, rotate, and display are operations that all subclasses inherit. Scale applies to one-dimensional and two-dimensional figures. Fill applies only to two-dimensional figures.

The word written next to the generalization line in the diagram—*dimensionality*—is a generalization set name. A generalization set name is an enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization. You should generalize only one aspect at a time. For example, the means of propulsion (wind, fuel, animal, gravity) and the operating environment (land, air, water, outer space) are two aspects for class Vehicle. Generalization set values are inherently in one-to-one correspondence with the subclasses of a generalization. The generalization set name is optional.
3.3 Generalization and Inheritance

class. Each subclass not only inherits all the features of its ancestors but adds its own specific features as well. For example, Pump adds attributes suctionPressure, dischargePressure, and flowRate, which other kinds of equipment do not share.

Figure 3.25 shows classes of geometric figures. This example has more of a programming flavor and emphasizes inheritance of operations. Move, select, rotate, and display are operations that all subclasses inherit. Scale applies to one-dimensional and two-dimensional figures. Fill applies only to two-dimensional figures.

Figure 3.25 Inheritance for graphic figures. Each subclass inherits the attributes, operations, and associations of its superclasses.

The word written next to the generalization line in the diagram—dimensionality—is a generalization set name. A generalization set name is an enumerated attribute that indicates which aspect of an object is being abstracted by a particular generalization. You should generalize only one aspect at a time. For example, the means of propulsion (wind, fuel, animal, gravity) and the operating environment (land, air, water, outer space) are two aspects for class Vehicle. Generalization set values are inherently in one-to-one correspondence with the subclasses of a generalization. The generalization set name is optional.
Do not nest subclasses too deeply. Deeply nested subclasses can be difficult to understand, much like deeply nested blocks of code in a procedural language. Often, with some careful thought and a little restructuring, you can reduce the depth of an overextended inheritance hierarchy. In practice, whether or not a subclass is "too deeply nested" depends upon judgment and the particular details of a problem. The following guidelines may help: An inheritance hierarchy that is two or three levels deep is certainly acceptable; ten levels deep is probably excessive; five or six levels may or may not be proper.

3.3.2 Use of Generalization

Generalization has three purposes, one of which is support for polymorphism. You can call an operation at the superclass level, and the OO language compiler automatically resolves the call to the method that matches the calling object's class. Polymorphism increases the flexibility of software—you add a new subclass and automatically inherit superclass behavior. Furthermore, the new subclass does not disrupt existing code. Contrast the OO situation with procedural code, where addition of a new type can cause a ripple of changes.

The second purpose of generalization is to structure the description of objects. When you use generalization, you are making a conceptual statement—you are forming a taxonomy and organizing objects on the basis of their similarities and differences. This is much more profound than modeling each class individually and in isolation from other classes.

The third purpose is to enable reuse of code—you can inherit code within your application as well as from past work (such as a class library). Reuse is more productive than repeatedly writing code from scratch. Generalization also lets you adjust the code, where necessary, to get the precise desired behavior. Reuse is an important motivator for inheritance, but the benefits are often oversold as Chapter 14 explains.

The terms generalization, specialization, and inheritance all refer to aspects of the same idea. Generalization and specialization concern a relationship among classes and take opposite perspectives, viewed from the superclass or from the subclasses. The word generalization derives from the fact that the superclass generalizes the subclasses. Specialization refers to the fact that the subclasses refine or specialize the superclass. Inheritance is the mechanism for sharing attributes, operations, and associations via the generalization/specialization relationship. In practice, there is little danger of confusion between the terms.

3.3.3 Overriding Features

A subclass may override a superclass feature by defining a feature with the same name. The overriding feature (the subclass feature) refines and replaces the overridden feature (the superclass feature). There are several reasons why you may wish to override a feature: to specify behavior that depends on the subclass, to tighten the specification of a feature, or to improve performance. For example, in Figure 3.25, each leaf subclass must implement display, even though Figure defines it. Class Circle improves performance by overriding operation rotate to be a null operation.

You may override methods and default values of attributes. You should never override the signature, or form, of a feature. An override should preserve attribute type, number and type of arguments to an operation, and operation return type. Tightening the type of an attribute or operation argument to be a subclass of the original type is a form of restriction and must be done with care. It is common to boost performance by overriding a general method with a special method that takes advantage of specific information but does not alter the operation semantics (such as Circle.rotate in Figure 3.25).

You should never override a feature so that it is inconsistent with the original inherited feature. A subclass is a special case of its superclass and should be compatible with it in every respect. A common, but unfortunate, practice in OO programming is to "borrow" a class that is similar to a desired class and then modify it by changing and ignoring some of its features, even though the new class is not really a special case of the original class. This practice can lead to conceptual confusion and hidden assumptions built into programs.

3.4 A Sample Class Model

Figure 3.26 shows a class model of a workstation window management system. This model is greatly simplified—a real model would require a number of pages—but it illustrates many class modeling constructs and shows how they fit together.

Class Window defines common parameters of all kinds of windows, including a rectangular boundary defined by the attributes x1, y1, x2, y2, and operations to display and undisplay a window and to raise it to the top (foreground) or lower it to the bottom (background) of the entire set of windows.

A canvas is a region for drawing graphics. It inherits the window boundary from Window and adds the dimensions of the underlying canvas region defined by attributes cx1, cx2, cy1, cy2. A canvas contains a set of elements, shown by the association to class Shape. All shapes have color and line width. Shapes can be lines, ellipses, or polygons, each with their own parameters. A polygon consists of a list of vertices. Ellipses and polygons are both closed shapes, which have a fill color and a fill pattern. Lines are one dimensional and cannot be filled. Canvas windows have operations to add and delete elements.

TextWindow is a kind of a ScrollingWindow which has a two-dimensional scrolling offset within its window, as specified by xoffset and yoffset, as well as an operation scroll to change the scroll value. A text window contains a string and has operations to insert and delete characters. ScrollingCanvas is a special kind of canvas that supports scrolling; it is both a Canvas and a ScrollingWindow. This is an example of multiple inheritance, to be explained in Chapter 4.

A Panel contains a set of PanelItem objects, each identified by a unique itemName within in a given panel, as shown by the qualified association. Each panel item belongs to a single panel. A panel item is a predefined icon with which a user can interact on the screen. Panel items come in three kinds: buttons, choice items, and text items. A button has a string that appears on the screen; a button can be pushed by the user and has an attribute depressed. A choice item allows the user to select one of a set of predefined choices, each of which is a ChoiceEntry containing a string to be displayed and a value to be returned if the entry is selected. There are two associations between ChoiceItem and ChoiceEntry; a one-to-many as-
association defines the set of allowable choices, while a one-to-one association identifies the current choice. The current choice must be one of the allowable choices, so one association is a subset of the other as shown by the arrow between them labeled "[subset]." This is an example of a constraint, to be explained in Chapter 4.

When a panel item is selected by the user, it generates an Event, which is a signal that something has happened together with an action to be performed. All kinds of panel items have notifyEvent associations. Each panel item has a single event, but one event can be shared among many panel items. Text items have a second kind of event, which is generated when a keyboard character is typed while the text item is selected. The association with end name keyboardEvent shows these events. Text items also inherit the notifyEvent from superclass PanelItem; the notifyEvent is generated when the entire text item is selected with a mouse.

There are many deficiencies in this model. For example, perhaps we should define a type Rectangle, which can then be used for the window and canvas boundaries, rather than having two similar sets of four position attributes. Maybe a line should be a special case of a polyline (a connected series of line segments), in which case both Polyline and Polygon could be subclasses of a new superclass that defines a list of points. Many attributes, operations, and classes are missing from a description of a realistic windowing system. Certainly the windows have associations among themselves, such as overlapping one another. Nevertheless, this simple model gives a flavor of the use of class modeling. We can criticize its details because it says something precise. It would serve as the basis for a fuller model.

### 3.5 Navigation of Class Models

So far we have shown how class models can express the structure of an application. Now we show how they can also express the behavior of navigating among classes. Navigation is important because it lets you exercise a model and uncover hidden flaws and omissions so that you can repair them. You can perform navigation manually (an informal technique) or write navigation expressions (as we will explain).

Consider the simple model for credit card accounts in Figure 3.27. An institution may issue many credit card accounts, each identified by an account number. Each account has a maximum credit limit, a current balance, and a mailing address. The account serves one or more customers who reside at the mailing address. The institution periodically issues a statement for each account. The statement lists a payment due date, finance charge, and minimum payment. The statement itemizes various transactions that have occurred throughout the billing interval: cash advances, interest charges, purchases, fees, and adjustments to the account. The name of the merchant is printed for each purchase.

We can pose a variety of questions against the model.

- What transactions occurred for a credit card account within a time interval?
- What volume of transactions were handled by an institution in the last year?
- What customers patronized a merchant in the last year by any kind of credit card?
How many credit card accounts does a customer currently have?

- What is the total maximum credit for a customer, for all accounts?

The UML incorporates a language that can express these kinds of questions—the Object Constraint Language (OCL) [Warner-99]. The next two sections discuss the OCL, and Section 3.5.3 then expresses the credit card questions using the OCL. By no means do we cover the complete OCL; we just cover the portions relevant to traversing class models.

### 3.5.1 OCL Constructs for Traversing Class Models

The OCL can traverse the constructs in class models.

- **Attributes.** You can traverse from an object to an attribute value. The syntax is the source object, followed by a dot, and then the attribute name. For example, the expression `aCreditCardAccount.maximumCredit` takes a `CreditCardAccount` object and finds the value of `maximumCredit` (We use the convention of preceding a class name by “a” to refer to an object.) Similarly, you can access an attribute for each object in a collection, returning a collection of attribute values. In addition, you can find an attribute value for a link, or a collection of attribute values for a collection of links.

- **Operations.** You can also invoke an operation for an object or a collection of objects. The syntax is the source object or object collection, followed by a dot, and then the operation. An operation must be followed by parentheses, even if it has no arguments, to avoid confusion with attributes. You may invoke operations from your class model or predefined operations that are built into the OCL.

The OCL has special operations that operate on entire collections (as opposed to operating on each object in a collection). For example, you can count the objects in a collection or sum a collection of numeric values. The syntax for a collection operation is the source object collection, followed by “." , and then the operation.

- **Simple associations.** A third use of the `.` notation is to traverse an association to a target end. The target end may be indicated by an association end name or, where there is no ambiguity, a class name. In the example, `aCustomer.MailingAddress` yields a set of addresses for a customer (the target end has “many” multiplicity). In contrast, `aCreditCardAccount.MailingAddress` yields a single address (the target end has multiplicity of one).

- **Qualified associations.** A qualifier lets you make a more precise traversal. The expression `aCreditCardAccount.Statement[30November1999]` finds the statement for a credit card account with the statement date of 30 November 1999. The syntax is to enclose the qualifier value in brackets. Alternatively, you can ignore the qualifier and traverse a qualified association as if it were a simple association. Thus the expression `aCreditCardAccount.Statement` finds the multiple statements for a credit card account. (The multiplicity is “many” when the qualifier is not used.)

- **Association classes.** Given a link of an association class, you can find the constituent objects. Alternatively, given a constituent object, you can find the multiple links of an association class.

- **Generalizations.** Traversal of a generalization hierarchy is implicit for the OCL notation.

- **Filters.** There is often a need to filter the objects in a set. The OCL has several kinds of filters, the most common of which is the `select` operation. The `select` operation applies a predicate to each element in a collection and returns the elements that satisfy the predicate. For example, `aStatement.Transaction->select(amount>$100)` finds the transactions for a statement in excess of $100.

### 3.5.2 Building OCL Expressions

The real power of the OCL comes from combining primitive constructs into expressions. For example, an OCL expression could chain together several association traversals. There could be several qualifiers, filters, and operators as well.

With the OCL, a traversal from an object through a single association yields a singleton or a set (or a bag if the association has the annotation [bag] or [sequence]). In general, a traversal through multiple associations can yield a bag (depending on the multiplicities), so you must be careful with OCL expressions. A set is a collection of elements without duplicates. A bag is a collection of elements with duplicates allowed.

The example in Figure 3.28 illustrates how an OCL expression can yield a bag. A company might want to send a single mailing to each stockholder address. Starting with the GE company, we traverse the `OwnsStock` association and get a set of three persons. Starting with
Figure 3.28 A sample model and examples. Traversal of multiple associations can yield a bag.

these three persons and traversing to mailing address, we get a bag obtaining the address 456 State twice.

[Warner-99] does not mention null values, since they only discuss the specification of constraints for a correctly implemented system. (Null is a special value denoting that an attribute value is unknown or not applicable.) Handling of exceptions and run-time errors is also outside the scope of their book.

In contrast, the purpose in this chapter is not to specify constraints, but rather to discuss navigation of class models. Nulls do not arise for properly phrased and valid constraints. But they certainly do arise with model navigation. For example, a person may lack a mailing address. We extend the meaning of OCL expressions to accommodate nulls—a traversal may yield a null value, and an OCL expression evaluates to null if the source object is null.

3.5.3 Examples of OCL Expressions

We can use the OCL to answer the credit card questions.

■ What transactions occurred for a credit card account within a time interval?

\[
\text{aCreditCardAccount.Statement.Transaction->}
\text{select(aStartDate <= transactionDate and transactionDate <= anEndDate)}
\]

The expression traverses from a CreditCardAccount object to Statement and then to Transaction, resulting in a set of transactions. (Traversal of the two associations results in a set, rather than a bag, because both associations are one-to-many.) Then we use the OCL select operator (a collection operator) to find the transactions within the time interval bounded by aStart Date and anEndDate.

■ What volume of transactions were handled by an institution in the last year?

\[
\text{anInstitution.CreditCardAccount.Statement.Transaction->}
\text{select(aStartDate <= transactionDate and transactionDate <= anEndDate).amount->sum()}
\]

The expression traverses from an Institution object to CreditCardAccount, then to Statement, and then to Transaction. (Traversal results in a set, rather than a bag, because all three associations are one-to-many.) The OCL select operator finds the transactions within the time interval bounded by aStartDate and anEndDate. (We choose to make the time interval more general than last year.) Then we find the amount for each transaction and compute the total with the OCL sum operator (a collection operator).

■ What customers patronized a merchant in the last year by any kind of credit card?

\[
\text{aMerchant.Purchase->}
\text{select(aStartDate <= transactionDate and transactionDate <= anEndDate).Statement.}
\text{CreditCardAccount.MailingAddress.Customer->asSet()}
\]

The expression traverses from a Merchant object to Purchase. The OCL select operator finds the transactions within the time interval bounded by aStartDate and anEndDate. (Traversal across a generalization, from Purchase to Transaction, is implicit in the OCL.) For these transactions, we then traverse to Statement, then to CreditCardAccount, then to MailingAddress, and finally to Customer. The association from MailingAddress to Customer is many-to-many, so traversal to Customer yields a bag. The OCL asSet operator converts a bag of customers to a set of customers, resulting in our answer.

■ How many credit card accounts does a customer currently have?

\[
\text{aCustomer.MailingAddress.CreditCardAccount->size()}
\]

Given a Customer object, we find a set of MailingAddress objects. Then, given the set of MailingAddress objects, we find a set of CreditCardAccount objects. (This traversal yields a set, and not a bag, because each CreditCardAccount pertains to a single MailingAddress.) For the set of CreditCardAccount objects we apply the OCL size operator, which returns the cardinality of the set.

■ What is the total maximum credit for a customer, for all accounts?

\[
\text{aCustomer.MailingAddress.CreditCardAccount.}
\text{maximumCredit->sum()}
\]

The expression traverses from a Customer object to MailingAddress, and then to CreditCardAccount, yielding a set of CreditCardAccount objects. For each CreditCardAccount, we find the value of maximumCredit and compute the total with the OCL sum operator.

Note that these kinds of questions exercise a model and uncover hidden flaws and omissions that can then be repaired. For example, the query on the number of credit card accounts suggests that we may need to differentiate past accounts from current accounts.

Keep in mind that the OCL was originally intended as a constraint language (see Chapter 4). However, as we explain here, the OCL is also useful for navigating models.
3.6 Practical Tips

We have gleaned the following tips for constructing class models from our application work. Many of these tips have been mentioned throughout the chapter.

- **Scope.** Don’t begin class modeling by merely jotting down classes, associations, and inheritance. First, you must understand the problem to be solved. The content of a model is driven by relevance to the solution. You must exercise judgment in deciding which objects to show and which objects to ignore. A model represents only the relevant aspects of a problem. (Section 3.1.1)

- **Simplicity.** Strive to keep your models simple. A simple model is easier to understand and takes less development effort. Try to use a minimal number of classes that are clearly defined and not redundant. Be suspicious of classes that are difficult to define. You may need to reconsider such classes and restructure the model.

- **Diagram layout.** Draw your diagrams in a manner that elicits symmetry. Often there is a superstructure to a problem that lies outside the notation. Try to position important classes so that they are visually prominent on a diagram. Try to avoid crossing lines.

- **Names.** Carefully choose names. Names are important and carry powerful connotations. Names should be descriptive, crisp, and unambiguous. Do not bias names toward one aspect of an object. Choosing good names is one of the most difficult aspects of modeling. You should use singular nouns for the names of classes.

- **References.** Do not bury object references inside objects as attributes. Instead, model these as associations. This is clearer and captures the true intent rather than an implementation approach. (Section 3.2.1)

- **Multiplicity.** Challenge association ends with a multiplicity of one. Often the object on either end is optional and zero-or-one multiplicity may be more appropriate. Other times “many” multiplicity is needed. (Section 3.2.2)

- **Association end names.** Be alert for multiple uses of the same class. Use association end names to unify references to the same class. (Section 3.2.3)

- **Bags and sequences.** An ordinary binary association has at most one link for a pair of objects. However, you can permit multiple links for a pair of objects by annotating an association end with [bag] or [sequence]. (Section 3.2.5)

- **Attributes of associations.** During analysis, do not collapse attributes of associations into one of the related classes. You should directly describe the objects and links in your models. During design and implementation, you can always combine information for more efficient execution. (Section 3.2.6)

- **Qualified associations.** Challenge association ends with a multiplicity of “many.” A qualifier can often improve the precision of an association and highlight important navigation paths. (Section 3.2.7)

- **Generalization levels.** Try to avoid deeply nested generalizations. (Section 3.3.1)

3.7 Chapter Summary

Class models describe the static data structure of objects and their relationships to one another. The content of a model is a matter of judgment and is driven by the needs of an application. An object is a concept, abstraction, or thing with identity that has meaning for an application. A class describes a group of objects with the same attributes, behavior, kinds of relationships, and semantics. An attribute is a named property of a class that describes a value held by each object of the class. An operation is a function or procedure that may be applied to or by objects in a class.

A link is a physical or conceptual connection among objects and is an instance of an association. An association is a description of a group of links with common structure and semantics. An association describes a set of potential links in the same way that a class describes a set of potential objects. An association is a logical construct, of which a reference is an implementation alternative. There are other ways of implementing associations besides using references.

You can refer to an end of an association and give it a name and multiplicity. Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. An association class is an association that is also a class; an association class may have attributes, operations, and participate in associations. A qualified association is an association in which the objects in a “many” association end are partially or fully disambiguated by an attribute called the qualifier. The qualifier selects among the target objects, reducing the effective multiplicity, often from “many” to “one.” Names are often qualifiers.

Generalization is the relationship between a class (the superclass) and one or more variations of the class (the subclasses). Generalization organizes classes by their similarities and differences, structuring the description of objects. A subclass inherits the attributes, operations, and associations of its superclasses. Through inheritance, a subclass can reuse superclass properties or override them; a subclass can add new properties.

Generalization is an important construct for both conceptual modeling and implementation. During conceptual modeling, generalization lets the developer organize classes on the basis of similarities and differences. During implementation, inheritance facilitates polymorphism and code reuse. Inheritance may occur across an arbitrary number of levels, where

- **Overriding features.** You may override methods and default values of attributes. However, you should never override a feature so that it is inconsistent with the signature or semantics of the original inherited feature. (Section 3.3.3)

- **Reviews.** Try to get others to review your models. Expect that your models will require revision. Class models require revision to clarify names, improve abstraction, repair errors, add information, and more accurately capture structural constraints. Nearly all of our models have required several revisions.

- **Documentation.** Always document your models. The diagram specifies the structure of a model but cannot describe the rationale. The written explanation guides the reader and explains subtle reasons for why the model was constructed a particular way.
each level represents one aspect of an object. An object accumulates attributes, operations, and associations from each level of a generalization hierarchy.

Class models are useful for more than just data structure. In particular, navigation of class models lets you express certain behavior. Furthermore, navigation exercises a class model and uncovers hidden flaws and omissions, which you can then repair. The UML incorporates a language that can be used for navigation, the Object Constraint Language (OCL).

The various class modeling constructs work together to describe a complex system precisely, as shown by our example of a model for a windowing system. Once a model is available, even a simplified one, you can compare it against the requirements of an application, criticize it, and improve it.

<table>
<thead>
<tr>
<th>concept</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ancestor</td>
<td>default value</td>
</tr>
<tr>
<td>descendant</td>
<td>link</td>
</tr>
<tr>
<td>direction</td>
<td>method</td>
</tr>
<tr>
<td>feature</td>
<td>qualified association</td>
</tr>
<tr>
<td>type</td>
<td>navigation</td>
</tr>
<tr>
<td>object</td>
<td>sequence</td>
</tr>
<tr>
<td>bag</td>
<td>signature</td>
</tr>
<tr>
<td>generalization</td>
<td>specialization</td>
</tr>
<tr>
<td>class diagram</td>
<td>inheriance</td>
</tr>
<tr>
<td>class model</td>
<td>override</td>
</tr>
<tr>
<td></td>
<td>value</td>
</tr>
</tbody>
</table>

Figure 3.29 Key concepts for Chapter 3

Bibliographic Notes

The class modeling approach described in this book builds on the OMT notation originally proposed in [Loomis-87], which has now been superseded by the UML [Booch-99] [Rumbaugh-05] [UML]. The UML class model corresponds to the OMT notation discussed in [Loomis-87]. [Blaha-98] also covers the UML class modeling notation with an emphasis on the constructs that are relevant to database applications.

The class modeling notation is one of a score of approaches descended from the seminal entity-relationship (ER) model of [Chen-76]. All the descendants attempt to improve on the ER approach. Enhancements to the ER model have been pursued for several reasons. The ER technique has been successful for database modeling and as a result, there has been great demand for additional power. Also, ER modeling addresses only database design and not programming. There are too many extensions to ER for us to discuss them here.

A noteworthy aspect of the OMT notation and its successor UML is the emphasis on associations. With inheritance, associations are important for conceptual modeling and implementation. [Rumbaugh-87] is the original source of the association ideas. The use of the term relation in [Rumbaugh-87] is synonymous with our use of association in this book.

In the data modeling notations, such as ER and IDEFI, a binary association has at most one link for a pair of objects. UML1 follows the data modeling convention and also restricts a binary association to at most one link for a pair of objects. Note that UML2 has an exception to this behavior. In UML2 a binary association with the annotation (bag) or (sequence) can have multiple links for a pair of objects.

[Khoshaian-86] defines the concept of object identity and its importance to programming languages and database systems. [Warner-99] is the reference for the Object Constraint Language (OCL) that is part of the UML. We use the OCL in this chapter for navigating class models.

[Rayside-00] compares OO concepts with philosophy. He emphasizes the importance of crisp names and clear thinking.

[Chonoles-03], [Fowler-00], and [Larman-02] are additional books that you can read to help you learn about the UML. We thank Michael Chonoles for the example (Figure 3.10, Figure 3.11) clarifying that each association has at most one link between a given pair of objects (other than bags and sequences).

References


[UML] www.uml.org

Exercises

3.1 (3) Prepare a class diagram from the object diagram in Figure E3.1.

```plaintext
<table>
<thead>
<tr>
<th>Borders</th>
<th>France: Country</th>
<th>Borders</th>
<th>Belgium: Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>xCoord=10</td>
<td>name=&quot;France&quot;</td>
<td>xCoord=10</td>
<td>name=&quot;Belgium&quot;</td>
</tr>
</tbody>
</table>
```

Figure E3.1 Object diagram for a portion of Europe

3.2 (5) Prepare a class diagram from the object diagram in Figure E3.2. Explain your multiplicity decisions. Each point has an x coordinate and a y coordinate. What is the smallest number of points required to construct a polygon? Does it make a difference whether or not a point may be shared between polygons? Your answer should address the fact that points are ordered.

```plaintext
<table>
<thead>
<tr>
<th>:Point</th>
<th>:Polygon</th>
</tr>
</thead>
<tbody>
<tr>
<td>xCoord=10</td>
<td>yCoord=10</td>
</tr>
</tbody>
</table>

Figure E3.2 Object diagram for a polygon that happens to be a square

3.3 (5) Using your class diagram for Exercise 3.2, prepare an object diagram for two triangles with a common side under the following conditions.

a. A point belongs to exactly one polygon.

b. A point belongs to one or more polygons.

3.4 (5) Prepare a class diagram from the object diagram in Figure E3.3. How does your diagram express the fact that points are ordered? Assume that a point belongs to at most one polygon.

3.5 (2) Prepare a written description for the class diagrams from Exercise 3.2 and Exercise 3.4.

3.6 (6) Prepare a class diagram from the object diagram in Figure E3.4.

3.7 (5) Prepare a class diagram from the object diagram in Figure E3.5. This particular document has 4 pages. The first page has a red point and a yellow square displayed on it. The second page contains a line and an ellipse. An arc, a circle, and a rectangle appear on the last two pages. In preparing your diagram, use one or more generalizations.

3.8 (4) Figure E3.6 is a partially completed class diagram of an air transportation system. Multiplicity has been omitted. Add multiplicity to the diagram. Demonstrate how multiplicity decisions depend on your perception of the world.

3.9 (3) Add association names to the unlabeled associations in Figure E3.6.

3.10 (3) Add association end names to Figure E3.6. Add only meaningful names that are different from the class names. You should add at least six association end names to the diagram.

3.11 (2) Add the following operations to the class diagram in Figure E3.6: heat, hire, fire, refuel, reserve, clean, de-ice, take off, land, repair, cancel, delay. It is permissible to add an operation to more than one class.

3.12 (6) Prepare an object diagram for an imaginary round trip you took last weekend to London. Include at least one instance of each class. Fortunately, direct flights on a hypersonic plane were available. A friend went with you but decided to stay a while and is still there. Captain Johnson was your pilot on both flights. You had a different seat each way, but you noticed it was on the same plane because of a distinctive dent in the tail section. Students should indicate unknown values with a '?'

3.13 Prepare a class diagram for each group of classes. Add at least 10 relationships (associations and generalizations) to each diagram. Use association names and association end names where
needed. Also use qualified associations and show multiplicity. You do not need to show attributes or operations. As you prepare the diagrams, you may add classes. Be sure to explain your diagrams.

a. (6) school, playground, principal, school board, classroom, book, student, teacher, cafeteria, restroom, computer, desk, chair, ruler, door, swing
b. (4) automobile, engine, wheel, brake, brake light, door, battery, muffler, tail pipe
c. (4) castle, moat, drawbridge, tower, ghost, stairs, dungeon, floor, corridor, room, window, stone, lord, lady, cook

d. (8) expression, constant, variable, function, argument list, relational operator, term, factor, arithmetic operator, statement, computer program
e. (6) file system, file, ASCII file, binary file, directory file, disc, drive, track, sector
f. (4) gas furnace, blower, blower motor, room thermostat, furnace thermostat, humidifier, humidity sensor, gas control, blower control, hot air vent
g. (7) chess piece, rank, file, square, board, move, tree of moves
h. (4) sink, freezer, refrigerator, table, light, switch, window, smoke alarm, burglar alarm, cabinet, bread, cheese, ice, door, kitchen

3.14 (4) Add at least 10 attributes and at least 5 methods to each of the class diagrams you prepared in the previous exercise.

3.15 (6) Figure E3.7 is a portion of a class diagram for a computer program for playing several types of card games. Deck, hand, discard pile, and draw pile are collections of cards. The initial size of a hand depends on the type of game. Each card has a suit and rank. Add the following operations to the diagram: display, shuffle, deal, initialize, sort, topOfPile, bottomOfPile, insert, draw, and discard. Some operations may appear in more than one class. For each class in which an operation appears, describe the arguments to the operation and what the operation should do to an instance of that class.
Modify the class diagram so that portions of the same column may appear on more than one page. If the user edits the text on one page, the changes should appear automatically on other pages. You should change x location and y location into attributes of an association.

3.17 (6) Figure E3.9 is a class diagram that might be used in developing a system to simplify the scheduling and scoring of judged athletic competitions such as gymnastics, diving, and figure skating. There are multiple events and competitors. Each competitor may enter several events and each event has many competitors.

Each event has several judges who subjectively rate the performance of competitors in that event. A judge rates every competitor for an event. In some cases, a judge may score more than one event.

Trials are the focus of the competition. Each trial is an attempt by one competitor to perform his or her best in one event. A trial is scored by the panel of judges for that event and a net score determined. Add multiplicity to the diagram.

![Diagram](image)

**Figure E3.9** Portion of a class diagram for an athletic-event scoring system

3.18 (3) Add the following attributes to Figure E3.9: address, age, date, difficulty factor, score, and name. In some cases, you may wish to use the same attribute in more than one class.

3.19 (3) Add an association to Figure E3.9 to make it possible to determine a competitor's intended events before trials are held.

3.20 (6) Prepare a class model to describe undirected graphs. An undirected graph consists of a set of vertices and a set of edges. Edges connect pairs of vertices. Your model should capture only the structure of graphs (i.e., connectivity) and need not be concerned with layout such as location of vertices or lengths of edges. Figure E3.10 shows a typical undirected graph.

![Diagram](image)

**Figure E3.10** Sample undirected graph

3.21 (4) Prepare an object diagram for Figure E3.10. [Instructor's note: You may want to give the students our answer to Exercise 3.20.]

3.22 (5) Extend the class diagram you prepared in Exercise 3.20 with layout details, including locations of vertices and thickness and color of edges. Also add names of vertices and edges. [Instructor's note: You may want to give the students our answer to Exercise 3.20.]

3.23 (7) Prepare a class model to describe directed graphs. A directed graph is similar to an undirected graph, except the edges are oriented. Figure E3.11 shows a typical directed graph.

![Diagram](image)

**Figure E3.11** Sample directed graph

3.24 (4) Prepare an object diagram for Figure E3.11. [Instructor's note: You may want to give the students our answer to Exercise 3.23.]

3.25 (7) Several classes shown in Figure E3.12 have attributes that are really references to other classes and could be replaced with associations. A person may have up to three companies as employers. Each person has an ID. A car is assigned an ID. Cars may be owned by persons, companies, or banks. Owner ID refers to the ID of the person, company, or bank who owns the car. A car loan may be involved in the purchase of a car.

Burying object references as references is the incorrect way to construct a model. Prepare a class diagram without IDs and using association and generalization. Try to assign multiplicities.

You may need to add one or more classes of your own.

<table>
<thead>
<tr>
<th>Person</th>
<th>Car</th>
<th>CarLoan</th>
<th>Company</th>
<th>Bank</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>name</td>
<td>name</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>birthdate</td>
<td></td>
<td></td>
<td>companyID</td>
<td></td>
</tr>
<tr>
<td>employer1ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>employer2ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>employer3ID</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>address</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ownerID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>vehicleID</td>
<td>customorType</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ownerType</td>
<td>model</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>accountNumber</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>banckID</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>interestRate</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>currentBalance</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Figure E3.12 | Classes with some attributes that are references. |

3.26 (4) A problem arises when several independent systems need to identify the same object. For example, the department of motor vehicles, an insurance company, a bank, and the police may wish to identify a given motor vehicle. Discuss the relative merits of using the following identification methods:

a. Identify by its owner
b. Identify by attributes such as manufacturer, model, and year
c. Use the vehicle identification number (VIN) assigned to the car by its manufacturer
d. Use IDs generated internally by each interested agency

3.27 (7) Prepare a class model that might be used to troubleshoot a 4-cycle lawn mower engine. Use three separate diagrams for the model, with one diagram for each of the following paragraphs.

Power is developed in such an engine by the combustion of a mixture of air and gasoline against a piston. The piston is attached to a crankshaft via a connecting rod, and it moves up and down inside a cylinder as the crank rotates. As the piston moves down, an intake valve opens, allowing the piston to draw a mixture of fuel and air into the cylinder. At the bottom of the stroke, the intake valve closes. The piston compresses and heats the mixture as it moves upward. Rings in grooves around the piston rub against the cylinder wall, providing a seal necessary for compression and spreading lubricating oil. At the top of the stroke, an electrical spark from a spark plug detonates the mixture. The expanding gases develop power during the downward stroke. At the bottom, an exhaust valve is opened. On the next upward stroke, the exhaust gases are driven out.

Fuel is mixed with air in a carburetor. Dust and dirt in the air, which could cause excessive mechanical wear, are removed by an air filter. The optimum ratio of fuel to air is set by adjusting a tapered mixture screw. A throttle plate controls the amount of mixture pulled into the cylinder. The throttle plate, in turn, is controlled through springs by the operator throttle control and a governor, a mechanical device which stabilizes the engine speed under varying mechanical loads. Intake and exhaust valves are normally held closed by springs and are opened at the right time by a cam shaft, which is gear driven by the crankshaft.

The electrical energy for the spark is provided and timed by a magnet, coil, condenser, and a normally closed switch called the points. The coil has a low-voltage primary circuit connected to the points and a high-voltage secondary circuit connected to the spark plug. The magnet is mounted on a flywheel and as it rotates past the coil, it induces a current in the shorted primary circuit. The points are driven open at the right instant by a cam on the crankshaft. With the aid of the condenser, they interrupt the current in the primary circuit, inducing a high-voltage pulse in the secondary.

3.28 (6) Prepare a class diagram for the dining philosopher problem. There are 5 philosophers and 5 forks around a circular table. Each philosopher has access to 2 forks, one on either side. Each fork is shared by 2 philosophers. Each fork may be either on the table or in use by one philosopher. A philosopher must have 2 forks to eat.

3.29 (7) The tower of Hanoi is a problem frequently used to teach recursive programming techniques. The goal is to move a stack of disks from one of three long pegs to another, using the third peg for maneuvering. Each disk is a different size. Disks may be moved from the top of a stack on a peg to the top of the stack on any other peg, one at a time, provided a disk is never placed on another disk that is smaller than itself. The details of the algorithm for solving the series of required moves depend on the structure of the class diagram used. Prepare class diagrams for each of the following descriptions. Show classes and associations. Do not show attributes or operations:

a. A tower consists of 3 pegs. Each peg has several disks on it, in a certain order.
b. A tower consists of 3 pegs. Disks on the pegs are organized into subsets called stacks. A stack is an ordered set of disks. Every disk is in exactly one stack. A peg may have several stacks on it, in order.
c. A tower consists of 3 pegs. Disks on the pegs are organized into subsets called stacks, as in (b), with several stacks on a peg. However, the structure of a stack is recursive. A stack con-

ists of one disk (the disk that is physically on the bottom of the stack) and zero or one stack, depending on the height of the stack.
d. Similar to (c), except only one stack is associated with a peg. Other stacks on the peg are associated in a linked list.

3.30 (8) The recursive algorithm for producing the series of moves described in the previous exercise focuses on a stack of disks. To move a stack of height $N$, where $N > 1$, first move the stack of height $N - 1$ to the free peg using a recursive call. Then move the bottom disk to the desired peg. Finally, move the stack on the free peg to the desired peg. The recursion terminates, because moving a stack of height 1 is trivial. Which one of the several class diagrams that you prepared in the previous exercise is best suited for this algorithm? Discuss why. Also, add attributes and operations to the diagram. What are the arguments for each operation? Describe what each operation is supposed to do to each class for which it is defined.

3.31 (6) Consider Figure E3.6. Write an OCL expression to compute the set of names of airlines that a person flew in a given year. Assume you have a function getYear(data) that extracts the year given a date. (Instructor’s note: You should give the students our answer to Exercise 3.10 as the basis for this exercise.)

3.32 (6) Consider Figure E3.6. Write an OCL expression to find the nonstop flights from aCity1 to aCity2. (Instructor’s note: You should give the students our answer to Exercise 3.10 as the basis for this exercise.)

3.33 (6) Consider Figure E3.9. Write an OCL expression to find the total score a competitor received from a judge. (Instructor’s note: You should give the students our answer to Exercise 3.18 as the basis for this exercise.)

3.34 (6) Compare the class models in Figure E3.13. The left model represents Subscription as an association class; the right model treats Subscription as an ordinary class.

A person may have multiple magazine subscriptions. A magazine has multiple subscribers. For each subscription, it is important to track the date and amount of each payment as well as the current expiration date.

(a) Person * subscriber Subscription * Magazine

(name address phone)

(expirationDate)

(b) Person * subscriber Subscription * Magazine

(name address phone)

(date amount)

Figure E3.13 Class diagram for magazine subscriptions
5

State Modeling

You can best understand a system by first examining its static structure—that is, the structure of its objects and their relationships to each other at a single moment in time (the class model). Then you should examine changes to the objects and their relationships over time (the state model). The state model describes the sequences of operations that occur in response to external stimuli, as opposed to what the operations do, what they operate on, or how they are implemented.

The state model consists of multiple state diagrams, one for each class with temporal behavior that is important to an application. The state diagram is a standard computer science concept (a graphical representation of finite state machines) that relates events and states. Events represent external stimuli and states represent values of objects. You should master the material in this chapter before proceeding in the book.

5.1 Events

An event is an occurrence at a point in time, such as user depresses left button or flight 123 departs from Chicago. Events often correspond to verbs in the past tense (power turned on, alarm set) or to the onset of some condition (paper tray becomes empty, temperature becomes lower than freezing). By definition, an event happens instantaneously with regard to the time scale of an application. Of course, nothing is really instantaneous; an event is simply an occurrence that an application considers atomic and fleeting. The time at which an event occurs is an implicit attribute of the event. Temporal phenomena that occur over an interval of time are properly modeled with a state.

One event may logically precede or follow another, or the two events may be unrelated. Flight 123 must depart Chicago before it can arrive in San Francisco; the two events are causally related. Flight 123 may depart before or after flight 456 departs Rome; the two events are causally unrelated. Two events that are causally unrelated are said to be concurrent; they have no effect on each other. If the communications delay between two locations exceeds the difference in event times, then the events must be concurrent because they cannot influence each other. Even if the physical locations of two events are not distant, we consider the events concurrent if they do not affect each other. In modeling a system we do not try to establish an ordering between concurrent events because they can occur in any order.

Events include error conditions as well as normal occurrences. For example, motor jammed, transaction aborted, and timeout are typical error events. There is nothing different about an error event; only our interpretation makes it an “error.”

The term event is often used ambiguously. Sometimes it refers to an instance, at other times to a class. In practice, this ambiguity is usually not a problem and the precise meaning is apparent from the context. If necessary, you can say event occurrence or event type to be precise.

There are several kinds of events. The most common are the signal event, the change event, and the time event.

5.1.1 Signal Event

A signal is an explicit one-way transmission of information from one object to another. It is different from a subroutine call that returns a value. An object sending a signal to another object may expect a reply, but the reply is a separate signal under the control of the second object, which may or may not choose to send it.

A signal event is the event of sending or receiving a signal. Usually we are more concerned about the receipt of a signal, because it causes effects in the receiving object. Note the difference between signal and signal event—a signal is a message between objects while a signal event is an occurrence in time.

Every signal transmission is a unique occurrence, but we group them into signal classes and give each signal class a name to indicate common structure and behavior. For example, UA flight 123 departs from Chicago on January 10, 1991 is an instance of signal class Flight-Departure. Some signals are simple occurrences, but most signal classes have attributes indicating the values they convey. For example, as Figure 5.1 shows, FlightDeparture has attributes airline, flightNumber, city, and date. The UML notation is the keyword signal in guillemets («») above the signal class name in the top section of a box. The second section lists the signal attributes.

![Figure 5.1 Signal classes and attributes](image)
5.1.2 Change Event

A change event is an event that is caused by the satisfaction of a boolean expression. The intent of a change event is that the expression is continually tested—whenever the expression changes from false to true, the event happens. Of course, an implementation would not continuously check a change event, but it must check often enough so that it seems continuous from an application perspective.

The UML notation for a change event is the keyword when followed by a parenthesized boolean expression. Figure 5.2 shows several examples of change events.

- when (room temperature < heating set point)
- when (room temperature > cooling set point)
- when (battery power < lower limit)
- when (tire pressure < minimum pressure)

Figure 5.2 Change events. A change event is an event that is caused by the satisfaction of a boolean expression.

5.1.3 Time Event

A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval. As Figure 5.3 shows, the UML notation for an absolute time is the keyword when followed by a parenthesized expression involving time. The notation for a time interval is the keyword after followed by a parenthesized expression that evaluates to a time duration.

- when (date = January 1, 2000)
- after (10 seconds)

Figure 5.3 Time events. A time event is an event caused by the occurrence of an absolute time or the elapse of a time interval.

5.2 States

A state is an abstraction of the values and links of an object. Sets of values and links are grouped together into a state according to the gross behavior of objects. For example, the state of a bank is either solvent or insolvent, depending on whether its assets exceed its liabilities. States often correspond to verbs with a suffix of “ing” (Waiting, Dialing) or the duration of some condition (Powered, BelowFreezing).

Figure 5.4 shows the UML notation for a state—a rounded box containing an optional state name. Our convention is to list the state name in boldface, center the name near the top of the box, and capitalize the first letter.

Figure 5.4 States. A state is an abstraction of the values and links of an object.

In defining states, we ignore attributes that do not affect the behavior of the object, and lump together in a single state all combinations of values and links with the same response to events. Of course, every attribute has some effect on behavior or it would be meaningless, but often some attributes do not affect the sequence of control and you can regard them as simple parameter values within a state. Recall that the purpose of modeling is to focus on qualities that are relevant to the solution of an application problem and abstract away those that are irrelevant. The three UML models (class, state, and interaction) present different views of a system for which the particular choice of attributes and values are not equally important. For example, except for leading 0s and 1s, the exact digits dialed do not affect the control of the phone line, so we can summarize them all with state Dialing and track the phone number as a parameter. Sometimes, all possible values of an attribute are important, but usually only when the number of possible values is small.

The objects in a class have a finite number of possible states—one or possibly some larger number. Each object can only be in one state at a time. Objects may parade through one or more states during their lifetime. At a given moment of time, the various objects for a class can exist in a multitude of states.

A state specifies the response of an object to input events. All events are ignored in a state, except those for which behavior is explicitly prescribed. The response may include the invocation of behavior or a change of state. For example, if a digit is dialed in state Dial tone, the phone line drops the dial tone and enters state Dialing; if the receiver is replaced in state Dial tone, the phone line goes dead and enters state Idle.

There is a certain symmetry between events and states as Figure 5.5 illustrates. Events represent points in time; states represent intervals of time. A state corresponds to the interval between two events received by an object. For example, after the receiver is lifted and before the first digit is dialed, the phone line is in state Dial tone. The state of an object depends on past events, which in most cases are eventually hidden by subsequent events. For example, events that happened before the phone is hung up do not affect future behavior; the Idle state "forgets" events received prior to the receipt of the hang up signal.

Figure 5.5 Event vs. state. Events represent points in time; states represent intervals of time.

Both events and states depend on the level of abstraction. For example, a travel agent planning an itinerary would treat each segment of a journey as a single event; a flight status
board in an airport would distinguish departures and arrivals; an air traffic control system would break each flight into many geographical legs.

You can characterize a state in various ways, as Figure 5.6 shows for the state Alarm ringing on a watch. The state has a suggestive name and a natural-language description of its purpose. The event sequence that leads to the state consists of setting the alarm, doing anything that doesn’t clear the alarm, and then having the target time occur. A declarative condition for the state is given in terms of parameters, such as current and target time; the alarm stops ringing after 20 seconds. Finally, a stimulus-response table shows the effect of events current time and button pushed, including the response that occurs and the next state. The different descriptions of a state may overlap.

<table>
<thead>
<tr>
<th>State: AlarmRinging</th>
</tr>
</thead>
<tbody>
<tr>
<td>Description: alarm on watch is ringing to indicate target time</td>
</tr>
<tr>
<td>Event sequence that produces the state:</td>
</tr>
<tr>
<td>setAlarm (targetTime)</td>
</tr>
<tr>
<td>when (currentTime = targetTime)</td>
</tr>
<tr>
<td>Condition that characterizes the state:</td>
</tr>
<tr>
<td>alarm = on, alarm set to targetTime, targetTime ≤ currentTime ≤ targetTime + 20 seconds, and no button has been pushed since targetTime</td>
</tr>
<tr>
<td>Events accepted in the state:</td>
</tr>
<tr>
<td>event</td>
</tr>
<tr>
<td>currentTime = targetTime + 20</td>
</tr>
<tr>
<td>buttonPushed (any button)</td>
</tr>
</tbody>
</table>

Figure 5.6 Various characterizations of a state. A state specifies the response of an object to input events.

Can links have state? In as much as they can be considered objects, links can have state. As a practical matter, it is generally sufficient to associate state only with objects.

5.3 Transitions and Conditions

A transition is an instantaneous change from one state to another. For example, when a called phone is answered, the phone line transitions from the Ringing state to the Connected state. The transition is said to fire upon the change from the source state to the target state. The origin and target of a transition usually are different states, but may be the same. A transition fires when its event occurs (unless an optional guard condition causes the event to be ignored). The choice of next state depends on both the original state and the event received.

An event may cause multiple objects to transition; from a conceptual point of view such transitions occur concurrently.

A guard condition is a boolean expression that must be true in order for a transition to occur. For example, a traffic light at an intersection may change only if a road has cars waiting. A guarded transition fires when its event occurs, but only if the guard condition is true. For example, “when you go out in the morning (event), if the temperature is below freezing (condition), then put on your gloves (next state).” A guard condition is checked only once, at the time the event occurs, and the transition fires if the condition is true. If the condition becomes true later, the transition does not then fire. Note that a guard condition is different from a change event—a guard condition is checked only once while a change event is, in effect, checked continuously.

Figure 5.7 shows guarded transitions for traffic lights at an intersection. One pair of electric eyes checks the north-south left turn lanes; another pair checks the east-west turn lanes. If no car is in the north-south and/or east-west turn lanes, then the traffic light control logic is smart enough to skip the left turn portion of the cycle.

![Guarded Transitions](image)

Figure 5.7 Guarded transitions. A transition is an instantaneous change from one state to another. A guard condition is a boolean expression that must be true in order for a transition to occur.

The UML notation for a transition is a line from the origin state to the target state. An arrowhead points to the target state. The line may consist of several line segments. An event may label the transition and be followed by an optional guard condition in square brackets. By convention, we usually confine line segments to a rectilinear grid. We italicize the event name and show the condition in normal font.

5.4 State Diagrams

A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states. A state diagram specifies the state sequences caused by event sequences. State names must be unique within the scope of a state diagram. All objects in a class execute the state diagram for that class, which models their common behavior. You can implement
state diagrams by direct interpretation or by converting the semantics into equivalent programming code.

The state model consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces—events and guard conditions. The individual state diagrams interact by passing events and through the side effects of guard conditions. Some events and guard conditions appear in a single state diagram; others appear in multiple state diagrams for the purpose of coordination. This chapter covers only individual state diagrams; Chapter 6 discusses state models of interacting diagrams.

A class with more than one state has important temporal behavior. Similarly, a class is temporally important if it has a single state with multiple responses to events. You can represent state diagrams with a single state in a simple nons graphical form—a stimulus—response table listing events and guard conditions and the ensuing behavior.

5.4.1 Sample State Diagram

Figure 5.8 shows a state diagram for a telephone line. The diagram concerns a phone line and not the caller or callee. The diagram contains sequences associated with normal calls as well as some abnormal sequences, such as timing out while dialing or getting busy lines. The UML notation for a state diagram is a rectangle with its name in a small pentagonal tag in the upper left corner. The constituent states and transitions lie within the rectangle.

At the start of a call, the telephone line is idle. When the phone is removed from the hook, it emits a dial tone and can accept the dialing of digits. Upon entry of a valid number, the phone system tries to connect the call and route it to the proper destination. The connection can fail if the number or trunk are busy. If the connection is successful, the called phone begins ringing. If the called party answers the phone, a conversation can occur. When the called party hangs up, the phone disconnects and reverts to idle when put on hook again.

Note that the receipt of the signal onHook causes a transition from any state to Idle (the bundle of transitions leading to Idle). Chapter 6 will show a more general notation that represents events applicable to groups of states with a single transition.

States do not totally define all values of an object. For example, state Dialing includes all sequences of incomplete phone numbers. It is not necessary to distinguish between different numbers as separate states, since they all have the same behavior, but the actual number dialed must of course be saved as an attribute.

If more than one transition leaves a state, then the first event to occur causes the corresponding transition to fire. If an event occurs and no transition matches it, then the event is ignored. If more than one transition matches an event, only one transition will fire, but the choice is nondeterministic.

5.4.2 One-shot State Diagrams

State diagrams can represent continuous loops or one-shot life cycles. The diagram for the phone line is a continuous loop. In describing ordinary usage of the phone, we do not know or care how the loop is started. (If we were describing installation of new lines, the initial state would be important.)

One-shot state diagrams represent objects with finite lives and have initial and final states. The initial state is entered on creation of an object; entry of the final state implies destruction of the object. Figure 5.9 shows a simplified life cycle of a chess game with a default initial state (solid circle) and a default final state (bull's eye).

As an alternate notation, you can indicate initial and final states via entry and exit points. In Figure 5.10 the start entry point leads to white's first turn, and the chess game eventually ends with one of three possible outcomes. Entry points (hollow circles) and exit points (circles enclosing an "x") appear on the state diagram's perimeter and may be named.
5.5 State Diagram Behavior

State diagrams would be of little use if they just described events. A full description of an object must specify what the object does in response to events.

5.5.1 Activity Effects

An effect is a reference to a behavior that is executed in response to an event. An activity is the actual behavior that can be invoked by any number of effects. For example, disconnectPhoneLine might be an activity that is executed in response to an onHook event for Figure 5.8. An activity may be performed upon a transition, upon the entry to or exit from a state, or upon some other event within a state.

Activities can also represent internal control operations, such as setting attributes or generating other events. Such activities have no real-world counterparts but instead are mechanisms for structuring control within an implementation. For example, a program might increment an internal counter every time a particular event occurs.

The notation for an activity is a slash ("/"") and the name (or description) of the activity, following the event that causes it. The keyword do is reserved for indicating an ongoing activity (to be explained) and may not be used as an event name. Figure 5.12 shows the state diagram for a pop-up menu on a workstation. When the right button is depressed, the menu is displayed; when the right button is released, the menu is erased. While the menu is visible, the highlighted menu item is updated whenever the cursor moves.
5.5.2 Do-Activities
A do-activity is an activity that continues for an extended time. By definition, a do-activity can only occur within a state and cannot be attached to a transition. For example, the warning light may flash during the Paper jam state for a copy machine (Figure 5.13). Do-activities include continuous operations, such as displaying a picture on a television screen, as well as sequential operations that terminate by themselves after an interval of time, such as closing a valve.

Figure 5.13 Do-activity for a copy machine. A do-activity is an activity that continues for an extended time.

The notation "do /" denotes a do-activity that may be performed for all or part of the duration that an object is in a state. A do-activity may be interrupted by an event that is received during its execution; such an event may or may not cause a transition out of the state containing the do-activity. For example, a robot moving a part may encounter resistance, causing it to cease moving.

5.5.3 Entry and Exit Activities
As an alternative to showing activities on transitions, you can bind activities to entry or to exit from a state. There is no difference in expressive power between the two notations, but frequently all transitions into a state perform the same activity, in which case it is more concise to attach the activity to the state.

For example, Figure 5.14 shows the control of a garage door opener. The user generates depress events with a pushbutton to open and close the door. Each event reverses the direction of the door, but for safety the door must open fully before it can be closed. The control generates motor up and motor down activities for the motor. The motor generates door open and door closed events when the motion has been completed. Both transitions entering state Opening cause the door to open.

Figure 5.14 Activities on transitions. An activity may be bound to an event that causes a transition.

Exit activities are less common than entry activities, but they are occasionally useful. An exit activity is shown inside the state box following the keyword exit and a "/" character. Whenever the state is exited, by any outgoing transition, the exit activity is performed first. If a state has multiple activities, they are performed in the following order: activities on the incoming transition, entry activities, do-activities, exit activities, activities on the outgoing transition. Events that cause transitions out of the state can interrupt do-activities. If a do-activity is interrupted, the exit activity is still performed.

In general, any event can occur within a state and cause an activity to be performed. Entry and exit are only two examples of events that can occur. As Figure 5.16 shows, there is a difference between an event within a state and a self-transition; only the self-transition causes the entry and exit activities to be executed.
5.5.4 Completion Transition

Often the sole purpose of a state is to perform a sequential activity. When the activity is completed, a transition to another state fires. An arrow without an event name indicates an automatic transition that fires when the activity associated with the source state is completed. Such unlabeled transitions are called completion transitions because they are triggered by the completion of activity in the source state.

A guard condition is tested only once, when the event occurs. If a state has one or more completion transitions, but none of the guard conditions are satisfied, then the state remains active and may become "stuck"—the completion event does not occur a second time, therefore no completion transition will fire later to change the state. If a state has completion transitions leaving it, normally the guard conditions should cover every possible outcome. You can use the special condition else to apply if all the other conditions are false. Do not use a guard condition on a completion transition to model waiting for a change of value. Instead model the waiting as a change event.

5.5.5 Sending Signals

An object can perform the activity of sending a signal to another object. A system of objects interacts by exchanging signals.

The activity "send target.(Attribute)" sends signal S with the given attributes to the target object or objects. For example, the phone line sends a connect(phone number) signal to the switcher when a complete phone number has been dialed. A signal can be directed at a set of objects or a single object. If the target is a set of objects, each of them receives a separate copy of the signal concurrently, and each of them independently processes the signal and determines whether to fire a transition. If the signal is always directed to the same object, the diagram can omit the target (but it must be supplied eventually in an implementation, of course).

If an object can receive signals from more than one object, the order in which concurrent signals are received may affect the final state; this is called a race condition. For example, in Figure 5.15 the door may or may not remain open if the button is pressed at about the time the door becomes fully open. A race condition is not necessarily a design error, but concurrent systems frequently contain unwanted race conditions that must be avoided by careful design. A requirement of two signals being received simultaneously is never a meaningful condition in the real world, as slight variations in transmission speed are inherent in any distributed system.

5.5.6 Sample State Diagram with Activities

Figure 5.17 adds activities to the state diagram from Figure 5.8.

5.6 Practical Tips

The precise content of all models depends on application needs. The chapter has already mentioned the following practical tips, and we summarize them here for your convenience.

- **Abstracting Values into States.** Consider only relevant attributes when defining a state. State diagrams need not use all attributes shown in a class model. (Section 5.2)
- **Parameters.** Parameterize events for incidental data that do not affect the flow of control. (Section 5.2)
- **Granularity of Events and States.** Consider application needs when deciding on the granularity of events and states. (Section 5.2)
- **When to Use State Diagrams.** Construct state diagrams only for classes with meaningful temporal behavior. A class has important temporal behavior if it responds differently to various events or has more than one state. Not all classes require a state diagram. (Section 5.4)
- **Entry and Exit Activities.** When a state has multiple incoming transitions, and all transitions cause the same activity to occur, use an entry activity within the state rather than repeatedly listing the activity on transition arcs. Do likewise for exit activities. (Section 5.5.3)
- **Guard Conditions.** Be careful with guard conditions so that an object does not become "stuck" in a state. (Section 5.5.4)
- **Race Conditions.** Beware of unwanted race conditions in state diagrams. Race conditions may occur when a state can accept events from more than one object. (Section 5.5.5)

5.7 Chapter Summary

Event and state are the two elementary concepts in state modeling. An event is an occurrence at a point in time. A state is an abstraction of the values and links of an object. Events represent points in time; states represent intervals of time. An object may respond to certain events when it is in certain states. All events are ignored in a state, except those for which behavior is explicitly prescribed. The same event can have different effects (or no effect) in different states.
A transition is an instantaneous change from one state to another and is caused by the occurrence of an event. An optional guard condition can cause the event to be ignored. A guard condition is a boolean expression that must be true in order for a transition to occur.

An effect is a reference to a behavior that is executed by objects in response to an event. An activity is the actual behavior that can be invoked by any number of effects. An activity may be performed upon a transition or upon an event within a state. A do-activity is an interruptible behavior that continues for an extended time. Consequently, a do-activity can occur only within a state and cannot be attached to a transition.

A state diagram is a graph whose nodes are states and whose directed arcs are transitions between states. A state diagram specifies the possible states, what transitions are allowed between states, what events cause the transitions to occur, and what behavior is executed in response to events. A state diagram describes the common behavior for the objects in a class; as each object has its own values and links, so too each object has its own state or position in the state diagram. The state model consists of multiple state diagrams, one state diagram for each class with important temporal behavior. The state diagrams must match on their interfaces—events and guard conditions.

Figure 5.18 Key concepts for Chapter 5

Bibliographic Notes

[Wieringa-98] has a thorough comparison of various ways for specifying software, including specification of the dynamic behavior of systems.

Finite state machines are a basic computer science concept and are described in any standard text on automata theory, such as [Hopcroft-01]. They are often described as recognizers or generators of formal languages. Basic finite state machines have limited expressive power. They have been extended with local variables and recursion as Augmented Transition Networks [Woods-70] and Recursive Transition Networks. These extensions expand the range of formal languages they can express but do little to address the combinatorial explosion that makes them unwieldy for practical control problems. (Chapter 6 addresses this.)

Traditional finite automata have been approached from a synchronous viewpoint. Petri nets [Reisig-92] formalize concurrency and synchronization of systems with distributed activity without resort to any notion of global time. Although they succeed well as an abstract conceptual model, they are too low-level and inexpressive to be useful for specifying large systems.
The need to specify interactive user interfaces has created several techniques for specifying control. This work is directed toward finding notations that clearly express powerful kinds of interactions while also being easily implementable. See [Green-86] for a comparison of some of these techniques.

The first edition of this book distinguished between actions (instantaneous behavior) and activities (lengthy behavior). UML2 has redefined both of these terms, and we have modified our explanation accordingly. UML2 now defines an activity as a specification of executable behavior and an action as a predefined primitive activity. In effect, the new definition of activity in UML2 subsumes the action and activity of the old book.

References


Exercises

5.1 (6) An extension ladder has a rope, pulley, and latch for raising, lowering, and locking the extension. When the latch is locked, the extension is mechanically supported and you may safely climb the ladder. To release the latch, you raise the extension slightly with the rope. You may then freely raise or lower the extension. The ladder produces a clacking sound as it passes over rungs of the ladder. The ladder may be reengaged while raising the extension by reversing direction just as the ladder is passing a rung. Prepare a state diagram of an extension ladder.

5.2 (4) A simple digital watch has a display and two buttons to set it, the A button and the B button. The watch has two modes of operation, display time and set time. In the display time mode, the watch displays hours and minutes, separated by a flashing colon.

The set time mode has two submodes, set hours and set minutes. The A button selects modes. Each time it is pressed, the mode advances in the sequence: display, set hours, set minutes, display, etc. Within the submodes, the B button advances the hours or minutes once each time it is pressed. Buttons must be released before they can generate another event. Prepare a state diagram of the watch.

5.3 (4) Figure E5.1 is a partially completed and simplified state diagram for the control of a telephone answering machine. The machine detects an incoming call on the first ring and answers the call with a prerecorded announcement. When the announcement is complete, the machine records the caller's message. When the caller hangs up, the machine hangs up and shuts off. Place the following in the diagram: call detected, answer play, announcement, record message, caller hangs up, announcement complete.

5.4 (7) The telephone answering machine in the previous exercise activates on the first ring. Revise the state diagram so that the machine answers after five rings. If someone answers the telephone before five rings, the machine should do nothing. Be careful to distinguish between five calls in which the telephone is answered on the first ring and one call that rings five times.

5.5 (3) In a personal computer: a disk controller is typically used to transfer a stream of bytes from a floppy disk drive to a memory buffer with the help of a host such as the central processing unit (CPU) or a direct memory access (DMA) controller. Figure E5.2 shows a partially completed and simplified state diagram for the control of the data transfer.

The controller signals the host each time a new byte is available. The data must then be read and stored before another byte is ready. When the disk controller senses the data has been read, it indicates that data is not available, in preparation for the next byte. If any byte is not read before the next one comes along, the disk controller asserts a data lost error signal until the disk controller is reset. Add the following to the diagram: reset, indicate data not available, indicate data available, data read by host, new data ready, indicate data lost.

5.6 (5) Figure E5.3 is a partially completed state diagram for one kind of motor control that is commonly used in household appliances. The separate appliance control determines when the motor should be on and continuously asserts on as an input to the motor control when the motor should be running.

When on is asserted, the motor control should start and run the motor. The motor starts by applying power to both the start and the run windings. A sensor, called a starting relay, determines when the motor has started, at which point the start winding is turned off, leaving only the run winding powered. Both windings are shut off when on is not asserted.

Appliance motors could be damaged by overheating if they are overloaded or fail to start. To protect against thermal damage, the motor control often includes an over-temperature sensor. If
Chapter 5 / State Modeling

5.7 (6) There was a single, continuously active input to the control in Exercise 5.6. In another common motor control, there are two pushbuttons, one for start and one for stop. To start the motor, the user presses the start button. The motor continues to run after the start button is released. To stop the motor, the user presses the stop button. The stop button takes precedence over the start button, so that the motor does not run while both buttons are pressed.

If both buttons are pressed and released, whether or not the motor starts depends on the order in which the buttons are released. If the stop button is released first, the motor starts. Otherwise the motor does not start. Modify the state diagram that you prepared in Exercise 5.6 to accommodate start and stop buttons.

5.8 (5) Prepare a state diagram for selecting and dragging objects with the diagram editor described in Exercise 4.2.

A cursor on the diagram tracks a two-button mouse. If the left button is pressed with the cursor on an object (a box or a line), the object is selected, replacing any previously selected object. If the left button is pressed with the cursor not on an object, the selection is set to null. Moving the mouse with the left button held down drags any selected object.

5.9 (6) Extend the diagram editor from Exercise 5.8. If the user left clicks on an object and holds the shift key, the object is added to the selection. Moving the mouse with the left button held down drags any selected objects.

5.10 (5) Figure E5.4 shows a state diagram for a copy machine. Initially the copy machine is off. When power is turned on, the machine reverts to a default state—one copy, automatic contrast, and normal size. While the machine is warming, it flashes the ready light. When the machine completes internal testing, the ready light stops flashing and remains on. Then the machine is ready for copying.

The operator may change any of the parameters when the machine is ready. The operator may increment or decrement the number of copies, change the size, toggle between automatic and manual contrast, and change the contrast when auto contrast is disabled. When the parameters are properly set, the operator presses the start button to begin making copies. Ordinarily, copying proceeds until all copies are made. Occasionally the machine may jam or run out of paper. When the machine jams, the operator may clear the blockage and the machine will resume making copies. Adding paper allows the machine to proceed after running out of paper.

Extend the diagram for the following observations. The copy machine does not work quite right. When it jams, the operator must first remove the jammed paper and then turn the machine off and on before it will operate correctly again. If the machine is turned off and on without first removing the offending paper, the machine stays jammed.

5.11 (7) While exploring an old castle, you and a friend discovered a bookcase that you suspected to be the entrance to a secret passageway. While you examined the bookcase, your friend removed a candle from its holder, only to discover that the candle holder was the entrance control. The bookcase rotated a half turn, pushing you along, separating you from your friend. Your friend put the candle back. This time the bookcase rotated a full turn, still leaving you behind it.

Your friend took the candle out. The bookcase started to rotate a full turn again, but this time you stopped it just shy of a full turn by blocking it with your body. Your friend handed you the candle and together you managed to force the bookcase back a half turn, but this left your friend behind it and you in front of it. You put the candle back. As the bookcase began to rotate, you took out the candle, and the bookcase stopped after a quarter turn. You and your friend then entered to explore further.

Prepare a state diagram for the control of the bookcase that is consistent with the previous scenario. What should you have done at first to gain entry with the least fuss?
(7) By adding states, extend the state diagram to accommodate another start-stop timer for a second channel.

d. (7) There is a great deal of commonality in your answer to the previous part. For example, setting the hour may be done in several contexts with similar results. Discuss how duplication of effort could be reduced.

6.9 (6) The diagram in Figure E5.4 has a major omission. The power can be turned off at any time, and the machine should transition to the off state. We could add a transition from each state to the off state, but this would clutter the diagram. Remedy this defect by using nested states.

6.10 (6) Figure E5.3 contains a class diagram for two persons playing a game of table tennis. Construct a state model corresponding to the class model.

The rules of table tennis are as follows. At the beginning of a game, the two players 'ping' for serve—that is, they bounce the ball over the net and hit it back and forth several times. The winner of the 'ping' serves first.

The winner of the 'ping' serves five times. Then the other player serves five times. Then the winner of the 'ping' serves five times again. This alternation of serve continues until the winner wins the game.

A game may be won upon shutdown (11-0) or when a player reaches 21 with at least a 2-point margin. If the score becomes tied at 20-20, the players then begin alternating individual serves until a player has a 2-point margin of victory.

![Class model for game of table tennis](image)

**Figure E6.3** Class model for game of table tennis

6.11 (10) Sometimes it is helpful to use reification—to promote something that is not an object into an object. Reification is a helpful technique for meta applications because it lets you shift the level of abstraction. On occasion it is useful to promote attributes, methods, constraints, and control information into objects so you can describe and manipulate them as data.

Construct a class model that reifies and supports the following state modeling concepts: event, state, transition, condition, activity, signal event, change event, and signal attribute.

6.12 (7) Take the model in Figure 6.5 and remove state nesting. That is, construct a flat state diagram with equivalent semantics.

## 7 Interaction Modeling

The interaction model is the third leg of the modeling tripod and describes interactions within a system. The class model describes the objects in a system and their relationships, the state model describes the life cycles of the objects, and the interaction model describes how the objects interact.

The interaction model describes how objects interact to produce useful results. It is a holistic view of behavior across many objects, whereas the state model is a reductionist view of behavior that examines each object individually. Both the state model and the interaction model are needed to describe behavior fully. They complement each other by viewing behavior from two different perspectives.

Interactions can be modeled at different levels of abstraction. At a high level, use cases describe how a system interacts with outside actors. Each use case represents a piece of functionality that a system provides to its users. Use cases are helpful for capturing informal requirements.

Sequence diagrams provide more detail and show the messages exchanged among a set of objects over time. Messages include both asynchronous signals and procedure calls. Sequence diagrams are good for showing the behavior seen by users of a system.

And finally, activity diagrams provide further detail and show the flow of control among the steps of a computation. Activity diagrams can show data flows as well as control flows. Activity diagrams document the steps necessary to implement an operation or a business process referenced in a sequence diagram.

### 7.1 Use Case Models

#### 7.1.1 Actors

An actor is a direct external user of a system—an object or set of objects that communicates directly with the system but that is not part of the system. Each actor represents those objects
that behave in a particular way toward the system. For example, customer and repair technician are different actors of a vending machine. For a travel agency system, actors might include traveler, agent, and airline. For a computer database system, actors might include user and administrator. Actors can be persons, devices, and other systems—anything that interacts directly with the system.

An object can be bound to multiple actors if it has different facets to its behavior. For example, the objects Mary, Frank, and Paul may be customers of a vending machine. Paul may also be a repair technician for the vending machine.

An actor has a single well-defined purpose. In contrast, objects and classes often combine many different purposes. An actor represents a particular facet of objects in its interaction with a system. The same actor can represent objects of different classes that interact similarly toward a system. For example, even though many different individuals use a vending machine, their behavior toward the vending machine can all be summarized by the actors customer and repair technician. Each actor represents a coherent set of capabilities for its objects.

Modeling the actors helps to define a system by identifying the objects within the system and those on its boundary. An actor is directly connected to the system—an indirectly connected object is not an actor and should not be included as part of the system model. Any interactions with an indirectly connected object must pass through the actors. For example, the dispatcher of repair technicians from a service bureau is not an actor of a vending machine—only the repair technician interacts directly with the machine. If it is necessary to model the interactions among such indirect objects, then a model should be constructed of the environment itself as a larger system. For example, it might be useful to build a model of a repair service that includes dispatchers, repair technicians, and vending machines as actors, but that is a different model from the vending machine model.

7.1.2 Use Cases

The various interactions of actors with a system are quantized into use cases. A use case is a coherent piece of functionality that a system can provide by interacting with actors. For example, a customer actor can buy a beverage from a vending machine. The customer inserts money into the machine, makes a selection, and ultimately receives a beverage. Similarly, a repair technician can perform scheduled maintenance on a vending machine. Figure 7.1 summarizes several use cases for a vending machine.

Each use case involves one or more actors as well as the system itself. The use case buy a beverage involves the customer actor and the use case perform scheduled maintenance involves the repair technician actor. In a telephone system, the use case make a call involves two actors, a caller and a receiver. The actors need not all be persons. The use case make a trade on an online stock broker involves a customer actor and a stock exchange actor. The stock broker system needs to communicate with both actors to execute a trade.

A use case involves a sequence of messages among the system and its actors. For example, in the buy a beverage use case, the customer first inserts a coin and the vending machine displays the amount deposited. This can be repeated several times. Then the customer pushes a button to indicate a selection; the vending machine dispenses the beverage and issues change, if necessary.

Some use cases have a fixed sequence of messages. More often, however, the message sequence may have some variations. For example, a customer can deposit a variable number of coins in the buy a beverage use case. Depending on the money inserted and the item selected, the machine may, or may not, return change. You can represent such variability by showing several examples of distinct behavior sequences. Typically you should first define a mainline behavior sequence, then define optional subsequences, repetitions, and other variations.

Error conditions are also part of a use case. For example, if the customer selects a beverage whose supply is exhausted, the vending machine displays a warning message. Similarly, the vending transaction can be cancelled. For example, the customer can push the coin return on the vending machine at any time before a selection has been accepted; the machine returns the coins, and the behavior sequence for the use case is complete. From the user’s point of view, some kinds of behavior may be thought of as errors. The designer, however, should plan for all possible behavior sequences. From the system’s point of view, user errors or resource failures are just additional kinds of behavior that a robust system can accommodate.

A use case brings together all of the behavior relevant to a slice of system functionality. This includes normal mainline behavior, variations on normal behavior, exception conditions, error conditions, and cancellations of a request. Figure 7.2 explains the buy a beverage use case in detail. Grouping normal and abnormal behavior under a single use case helps to ensure that all the consequences of an interaction are considered together.

In a complete model, the use cases partition the functionality of the system. They should preferably all be at a comparable level of abstraction. For example, the use cases make telephone call and record voice mail message are at comparable levels. The use case set external speaker volume to high is too narrow. It would be better as set speaker volume (with the volume level selection as part of the use case) or maybe even just set telephone parameters, under which we might group setting volume, display pad settings, setting the clock, and so on.
Use Case: Buy a beverage

Summary: The vending machine delivers a beverage after a customer selects and pays for it.

Actors: Customer

Preconditions: The machine is waiting for money to be inserted.

Description: The machine starts in the waiting state in which it displays the message “Enter coins.” A customer inserts coins into the machine. The machine displays the total value of money entered and lights up the buttons for the items that can be purchased for the money inserted. The customer presses a button. The machine dispenses the corresponding item and makes change, if the cost of the item is less than the money inserted.

Exceptions:

Canceled: If the customer presses the cancel button before an item has been selected, the customer’s money is returned and the machine resets to the waiting state.

Out of stock: If the customer presses a button for an out-of-stock item, the message “That item is out of stock” is displayed. The machine continues to accept coins or a selection.

Insufficient money: If the customer presses a button for an item that costs more than the money inserted, the message “You must insert $nn.nn more for that item” is displayed, where $nn.nn is the amount of additional money needed. The machine continues to accept coins or a selection.

No change: If the customer has inserted enough money to buy the item but the machine cannot make the correct change, the message “Cannot make correct change” is displayed and the machine continues to accept coins or a selection.

Postconditions: The machine is waiting for money to be inserted.

7.1.3 Use Case Diagrams

A system involves a set of use cases and a set of actors. Each use case represents a slice of the functionality the system provides. The set of use cases shows the complete functionality of the system at some level of detail. Similarly, each actor represents one kind of object for which the system can perform behavior. The set of actors represents the complete set of objects that the system can serve. Objects accumulate behavior from all the systems with which they interact as actors.

The UML has a graphical notation for summarizing use cases and Figure 7.3 shows an example. A rectangle contains the use cases for a system with the actors listed on the outside. The name of the system may be written near a side of the rectangle. A name within an ellipse denotes a use case. A “stick man” icon denotes an actor, with the name being placed below or adjacent to the icon. Solid lines connect use cases to participating actors.

In the figure, the actor Repair technician participates in two use cases, the others in one each. Multiple actors can participate in a use case, even though the example has only one actor per use case.

7.1.4 Guidelines for Use Case Models

Use cases identify the functionality of a system and organize it according to the perspective of users. In contrast, traditional requirements lists can include functionality that is vague to users, as well as overlook supporting functionality, such as initialization and termination. Use cases describe complete transactions and are therefore less likely to omit necessary steps. There is still a place for traditional requirements lists in describing global constraints and other nonlocalized functionality, such as mean time to failure and overall throughput, but you should capture most user interactions with use cases. The main purpose of a system is almost always found in the use cases, with requirements lists supplying additional implementation constraints. Here are some guidelines for constructing use case models:

- First determine the system boundary. It is impossible to identify use cases or actors if the system boundary is unclear.

- Ensure that actors are focused. Each actor should have a single, coherent purpose. If a real-world object embodies multiple purposes, capture them with separate actors. For example, the owner of a personal computer may install software, set up a database, and send email. These functions differ greatly in their impact on the computer system and the potential for system damage. They might be broken into three actors: system admin-
Each use case must provide value to users. A use case should represent a complete transaction that provides value to users and should not be defined too narrowly. For example, dial a telephone number is not a good use case for a telephone system. It does not represent a complete transaction of value by itself; it is merely part of the use case make telephone call. The latter use case involves placing the call, talking, and terminating the call. By dealing with complete use cases, we focus on the purpose of the functionality provided by the system, rather than jumping into implementation decisions. The details come later. Often there is more than one way to implement desired functionality.

Relate use cases and actors. Every use case should have at least one actor, and every actor should participate in at least one use case. A use case may involve several actors, and an actor may participate in several use cases.

Remember that use cases are informal. It is important not to be obsessed by formalism in specifying use cases. They are not intended as a formal mechanism but as a way to identify and organize system functionality from a user-centered point of view. It is acceptable if use cases are a bit loose at first. Detail can come later as use cases are expanded and mapped into implementations.

Use cases can be structured. For many applications, the individual use cases are completely distinct. For large systems, use cases can be built out of smaller fragments using relationships (see Chapter 8).

7.2 Sequence Models

The sequence model elaborates the themes of use cases. There are two kinds of sequence models: scenarios and a more structured format called sequence diagrams.

7.2.1 Scenarios

A scenario is a sequence of events that occurs during one particular execution of a system, such as for a use case. The scope of a scenario can vary; it may include all events in the system, or it may include only those events impinging on or generated by certain objects. A scenario can be the historical record of executing an actual system or a thought experiment of executing a proposed system.

A scenario can be displayed as a list of text statements, as Figure 7.4 illustrates. In this example, John Doe logs on with an online stock broker system, places an order for GE stock, and then logs off. Sometime later, after the order is executed, the securities exchange reports the results of the trade to the broker system. John Doe will see the results on the next login, but that is not part of this scenario.

The example expresses interaction at a high level. For example, the step John Doe logs in might require several messages between John Doe and the system. The essential purpose of the step, however, is the request to enter the system and providing the necessary identifi-

<table>
<thead>
<tr>
<th>John Doe logs in.</th>
</tr>
</thead>
<tbody>
<tr>
<td>System establishes secure communications.</td>
</tr>
<tr>
<td>System displays portfolio information.</td>
</tr>
<tr>
<td>John Doe enters a buy order for 100 shares of GE at the market price.</td>
</tr>
<tr>
<td>System verifies sufficient funds for purchase.</td>
</tr>
<tr>
<td>System displays confirmation screen with estimated cost.</td>
</tr>
<tr>
<td>John Doe confirms purchase.</td>
</tr>
<tr>
<td>System places order on securities exchange.</td>
</tr>
<tr>
<td>System displays transaction tracking number.</td>
</tr>
<tr>
<td>John Doe logs out.</td>
</tr>
<tr>
<td>System establishes insecure communication.</td>
</tr>
<tr>
<td>System displays good-bye screen.</td>
</tr>
<tr>
<td>Securities exchange reports results of trade.</td>
</tr>
</tbody>
</table>

Figure 7.4 Scenario for a session with an online stock broker. A scenario is a sequence of events that occurs during one particular execution of a system.

cation—the details can be shown separately. At early stages of development, you should express scenarios at a high level. At later stages, you can show the exact messages. Determining the detailed messages is part of development.

A scenario contains messages between objects as well as activities performed by objects. Each message transmits information from one object to another. For example, John Doe logs in transmits a message from John Doe to the broker system. The first step of writing a scenario is to identify the objects exchanging messages. Then you must determine the sender and receiver of each message, as well as the sequence of the messages. Finally, you can add activities for internal computations as scenarios are reduced to code.

7.2.2 Sequence Diagrams

A text format is convenient for writing, but it does not clearly show the sender and receiver of each message, especially if there are more than two objects. A sequence diagram shows the participants in an interaction and the sequence of messages among them. A sequence diagram shows the interaction of a system with its actors to perform all or part of a use case.

Figure 7.5 shows a sequence diagram corresponding to the previous stock broker scenario. Each actor as well as the system is represented by a vertical line called a lifeline and each message by a horizontal arrow from the sender to the receiver. Time proceeds from top to bottom, but the spacing is irrelevant; the diagram shows only the sequence of messages, not their exact timing. (Real-time systems impose time constraints on event sequences, but that requires extra notation.) Note that sequence diagrams can show concurrent signals—stock broker system sends messages to customer and securities exchange concurrently—and signals between participants need not alternate—stock broker system sends secure communication followed by display portfolio.

Each use case requires one or more sequence diagrams to describe its behavior. Each sequence diagram shows a particular behavior sequence of the use case. It is best to show a specific portion of a use case and not attempt to be too general. Although it is possible to
show conditionals within a sequence diagram, usually it is clearer to prepare one sequence diagram for each major flow of control.

Sequence diagrams can show large-scale interactions, such as an entire session with the stock broker system, but often such interactions contain many independent tasks that can be combined in various ways. Rather than repeating information, you can draw a separate sequence diagram for each task. For example, Figure 7.6 and Figure 7.7 show an order to purchase a stock and a request for a quote on a stock. These and various other tasks (not shown) would fit within an entire stock trading session.

You should also prepare a sequence diagram for each exception condition within the use case. For example, Figure 7.8 shows a variation in which the customer does not have sufficient funds to place the order. In this example, the customer cancels the order. In another variation (not shown), the customer would reduce the number of shares purchased and the order would be accepted.

In most systems, there are an unlimited number of scenarios, so it is not possible to show them all. However, you should try to elaborate all the use cases and cover the basic kinds of behavior with sequence diagrams. For example, a stock broker system can interleave purchases, sales, and inquiries arbitrarily. It is unnecessary to show all combinations of activities, once the basic pattern is established.
7.2.3 Guidelines for Sequence Models

The sequence model adds detail and elaborates the informal themes of use cases. There are two kinds of sequence models. Scenarios document a sequence of events with prose. Sequence diagrams also document the sequence of events but more clearly show the actors involved. The following guidelines will help you with sequence models.

- Prepare at least one scenario per use case. The steps in the scenario should be logical commands, not individual button clicks. Later, during implementation, you can specify the exact syntax of input. Start with the simplest mainline interaction—no repetitions, one main activity, and typical values for all parameters. If there are substantially different mainline interactions, write a scenario for each.
- Abstract the scenarios into sequence diagrams. The sequence diagrams clearly show the contribution of each actor. It is important to separate the contribution of each actor as a prelude to organizing behavior about objects.
- Divide complex interactions. Break large interactions into their constituent tasks and prepare a sequence diagram for each of them.
- Prepare a sequence diagram for each error condition. Show the system response to the error condition.

7.3 Activity Models

An activity diagram shows the sequence of steps that make up a complex process, such as an algorithm or workflow. An activity diagram shows flow of control, similar to a sequence diagram, but focuses on operations rather than on objects. Activity diagrams are most useful during the early stages of designing algorithms and workflows.

Figure 7.9 shows an activity diagram for the processing of a stock trade order that has been received by an online stock broker. The elongated ovals show activities and the arrows show their sequencing. The diamond shows a decision point and the heavy bar shows splitting or merging of concurrent threads.

The online stock broker first verifies the order against the customer’s account, then executes it with the stock exchange. If the order executes successfully, the system does three things concurrently: mails trade confirmation to the customer, updates the online portfolio to reflect the results of the trade, and settles the trade with the other party by debiting the account and transferring cash or securities. When all three concurrent threads have been completed, the system merges control into a single thread and closes the order. If the order execution fails, then the system sends a failure notice to the customer and closes the order.

An activity diagram is like a traditional flowchart in that it shows the flow of control from step to step. Unlike a traditional flowchart, however, an activity diagram can show both sequential and concurrent flow of control. This distinction is important for a distributed system. Activity diagrams are often used for modeling human organizations because they involve many objects—persons and organizational units—that perform operations concurrently.

Figure 7.9 Activity diagram for stock trade processing. An activity diagram shows the sequence of steps that make up a complex process.

7.3.1 Activities

The steps of an activity diagram are operations, specifically activities from the state model. The purpose of an activity diagram is to show the steps within a complex process and the sequencing constraints among them.

Some activities run forever until an outside event interrupts them, but most activities eventually complete their work and terminate by themselves. The completion of an activity is a completion event and usually indicates that the next activity can be started. An unlabeled arrow from one activity to another in an activity diagram indicates that the first activity must complete before the second activity can begin.

An activity may be decomposed into finer activities. For example, Figure 7.10 expands the execute order activity of Figure 7.9. It is important that the activities on a diagram be at the same level of detail. For example, in Figure 7.9 execute order and settle trade are similar in detail; they both express a high-level operation without showing the underlying mechanisms. If one of these activities were replaced in the activity diagram by its more detailed steps, the other activities should be replaced as well to maintain balance. Alternatively, balance can be preserved by elaborating the activities in separate diagrams.
arrow toward the first activities. A bull’s-eye (a solid circle surrounded by a hollow circle) shows the termination point—this symbol only has incoming arrows. When control reaches a bull’s-eye, the overall activity is complete and execution of the activity diagram ends.

### 7.3.4 Concurrent Activities

Unlike traditional flow charts, organizations and computer systems can perform more than one activity at a time. The pace of activity can also change over time. For example, one activity may be followed by another activity (sequential control), then split into several concurrent activities (a fork of control), and finally be combined into a single activity (a merge of control). A fork or merge is shown by a synchronization bar—a heavy line with one or more input arrows and one or more output arrows. On a synchronization, control must be present on all of the incoming activities, and control passes to all of the outgoing activities.

Figure 7.9 illustrates both a fork and merge of control. Once an order is executed, there is a fork—several tasks need to occur and they can occur in any order. The stock trade system must send confirmation to the customer, debit the customer’s account, and update the customer’s online portfolio. After the three concurrent tasks complete and the trade is settled, there is a merge, and execution proceeds to the activity of closing the order.

### 7.3.5 Executable Activity Diagrams

Activity diagrams are not only useful for defining the steps in a complex process, but they can also be used to show the progression of control during execution. An activity token can be placed on an activity symbol to indicate that it is executing. When an activity completes, the token is removed and placed on the outgoing arrow. In the simplest case, the token then moves to the next activity.

If there are multiple outgoing arrows with conditions, the conditions are examined to determine the successor activity. Only one successor activity can receive the token, even if more than one condition is true. If no condition is satisfied, the activity diagram is ill formed.

Multiple tokens can arise through concurrency. If an executing activity is followed by a concurrent split of control, completion causes an increase in the number of tokens—a token is placed on each of the concurrent activities. Similarly, a merge of control causes a decrease in the number of tokens as tokens migrate from the input activities to the output activities. All the input activities must complete before the merge can actually occur.

### 7.3.6 Guidelines for Activity Models

Activity diagrams elaborate the details of computation, thus documenting the steps needed to implement an operation or a business process. In addition, activity diagrams can help developers understand complex computations by graphically displaying the progression through intermediate execution steps. Here is some advice for activity models:

- **Don’t misuse activity diagrams.** Activity diagrams are intended to elaborate use case and sequence models so that a developer can study algorithms and workflow. Activity diagrams supplement the object-oriented focus of UML models and should not be used as an excuse to develop software via flowcharts.

---

**Figure 7.10 Activity diagram for execute order.** An activity may be decomposed into finer activities.

**7.3.2 Branches**

If there is more than one successor to an activity, each arrow may be labeled with a condition in square brackets, for example, [failure]. All subsequent conditions are tested when an activity completes. If one condition is satisfied, its arrow indicates the next activity to perform. If no condition is satisfied, the diagram is badly formed and the system will hang unless it is interrupted at some higher level. To avoid this danger, you can use the else condition; it is satisfied in case no other condition is satisfied. If multiple conditions are satisfied, only one successor activity executes, but there is no guarantee which one it will be. Sometimes this kind of nondeterminism is desirable, but often it indicates an error, so the modeler should determine whether any overlap of conditions can occur and whether it is correct.

As a notational convenience, a diamond shows a branch into multiple successors, but it means the same thing as arrows leaving an activity symbol directly. In Figure 7.8 the diamond has one incoming arrow and two outgoing arrows, each with a condition. A particular execution chooses only one path of control.

If several arrows enter an activity, the alternate execution paths merge. Alternatively, several arrows may enter a diamond and one may exit to indicate a merge.

**7.3.3 Initiation and Termination**

A solid circle with an outgoing arrow shows the starting point of an activity diagram. When an activity diagram is activated, control starts at the solid circle and proceeds via the outgoing
Chapter 7 / Interaction Modeling

- **Level diagrams.** Activities on a diagram should be at a consistent level of detail. Place additional detail for an activity in a separate diagram.

- **Be careful with branches and conditions.** If there are conditions, at least one must be satisfied when an activity completes—consider using an else condition. In undeterministic models, it is possible for multiple conditions to be satisfied—otherwise this is an error condition.

- **Be careful with concurrent activities.** Concurrency means that the activities can complete in any order and still yield an acceptable result. Before a merge can happen, all inputs must first complete.

- **Consider executable activity diagrams.** Executable activity diagrams can help developers understand their systems better. Sometimes they can even be helpful for end users who want to follow the progression of a process.

### 7.4 Chapter Summary

The interaction model provides a holistic view of behavior—how objects interact and exchange messages. At a high level, use cases partition the functionality of a system into discrete pieces meaningful to external actors. You can elaborate the behavior of use cases with scenarios and sequence diagrams. Sequence diagrams clearly show the objects in an interaction and the messages among them. Activity diagrams specify the details of a computation.

The class, state, and interaction models all involve the same concepts, namely data, sequencing, and operations, but each model focuses on a particular aspect and leaves the other aspects uninterpreted. All three models are necessary for a full understanding of a problem, although the balance of importance among the models varies according to the kind of application. The three models come together in the implementation of methods, which involve data (target object, arguments, and variables), control (sequencing constructs), and interactions (messages, calls, and sequences).

<table>
<thead>
<tr>
<th>activity</th>
<th>concurrency</th>
<th>scenario</th>
<th>use case</th>
</tr>
</thead>
<tbody>
<tr>
<td>activity diagram</td>
<td>interaction model</td>
<td>sequence diagram</td>
<td>use case diagram</td>
</tr>
<tr>
<td>activity token</td>
<td>lifeline</td>
<td>system boundary</td>
<td></td>
</tr>
<tr>
<td>actor</td>
<td>message</td>
<td>thread</td>
<td></td>
</tr>
</tbody>
</table>

Figure 7.11 Key concepts for Chapter 7

### Bibliographic Notes

Jacobson first introduced use cases [Jacobson-92]. The first edition of this book included scenarios and event trace diagrams. The latter are equivalent to simple sequence diagrams.

### References


### Exercises

#### 7.1 Consider a physical bookstore, such as in a shopping mall.

a. (2) List three actors that are involved in the design of a checkout system. Explain the relevance of each actor.

b. (2) One use case is the purchase of items. Take the perspective of a customer and list another use case at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.

c. (4) Prepare a use case diagram for a physical bookstore checkout system.

d. (3) Prepare a normal scenario for each use case. Remember that a scenario is an example, and need not exercise all functionality of the use case.

e. (3) Prepare an exception scenario for each use case.

f. (5) Prepare a sequence diagram corresponding to each scenario in (d).

#### 7.2 Consider a computer email system.

a. (2) List three actors. Explain the relevance of each actor.

b. (2) One use case is to get email. List four additional use cases at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.

c. (4) Prepare a use case diagram for a computer email system.

d. (3) Prepare a normal scenario for each use case. Remember that a scenario is an example, and need not exercise all functionality of the use case.

e. (3) Prepare an exception scenario for each use case.

f. (5) Prepare a sequence diagram corresponding to each scenario in (d).

#### 7.3 Consider an online airline reservation system. You may want to check airline Web sites to give you ideas.

a. (2) List two actors. Explain the relevance of each actor.

b. (2) One use case is to make a flight reservation. List four additional use cases at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.

c. (4) Prepare a use case diagram for an online airline reservation system.

#### 7.4 Consider a software system for supporting checkout of materials at a public library.

a. (2) List four actors. Explain the relevance of each actor.

b. (2) One use case is to borrow a library item. List three additional use cases at a comparable level of abstraction. Summarize the purpose of each use case with a sentence.

c. (4) Prepare a use case diagram for a library checkout system.

#### 7.5 (3) Identify at least 10 use cases for the Windows Explorer. Just list them textually and summarize the purpose of each use case in one or two sentences.

#### 7.6 (3) Write scenarios for the following situations:

a. Moving a bag of corn, a goose, and a fox across a river in a boat. Only one thing may be carried in the boat at a time. If the goose is left alone with the corn, the corn will be eaten. If
the goose is left alone with the fox, the goose will be eaten. Prepare two scenarios, one in which something gets eaten and one in which everything is safely transported across the river.
b. Getting ready to take a trip in your car. Assume an automatic transmission. Don’t forget your seat belt and emergency brake.
c. An elevator ride to the top floor.
d. Operation of a car cruise control. Include an encounter with slow-moving traffic that requires you to disengage and then resume control.

7.7 (4) Some combined bath–shower have two faucets and a lever for controlling the flow of the water. The lever controls whether the water flows from the shower head or directly into the tub. When the water is first turned on, it flows directly into the tub. When the lever is pulled, a valve closes and latches, diverting the flow of water to the shower head. To switch from shower to "bath with the water running, one must push the lever. Shutting off the water releases the lever, so that the next time the water is turned on, it flows directly into the tub. Write a scenario for a shower that is interrupted by a telephone call.

7.8 (4) Prepare an activity diagram for computing a restaurant bill. There should be a charge for each delivered item. The total amount should be subject to tax and a service charge of 18% for groups of six or more. For smaller groups, there should be a blank entry for a gratuity according to the customer’s discretion. Any coupons or gift certificates submitted by the customer should be subtracted.

7.9 (4) Prepare an activity diagram for awarding frequent flyer credits. In the past, TWA awarded a minimum of 750 miles for each flight. Gold and red card holders received a minimum of 1000 miles per flight. Gold card holders received a 25% bonus for any flight. Red card holders received a 50% bonus for any flight.

7.10 (5) Prepare an activity diagram that elaborates the details of logging into an email system. Note that entry of the user name and the password can occur in any order.

8 Advanced Interaction Modeling

The interaction model has several advanced features that can be helpful. You can skip this chapter on a first reading of the book.

8.1 Use Case Relationships

Independent use cases suffice for simple applications. However, it can be helpful to structure use cases for large applications. Complex use cases can be built from smaller pieces with the include, extend, and generalization relationships.

8.1.1 Include Relationship

The include relationship incorporates one use case within the behavior sequence of another use case. An included use case is like a subroutine—it represents behavior that would otherwise have to be described repeatedly. Often the fragment is a meaningful unit of behavior for the actors, although this is not required. The included use case may or may not be usable on its own.

The UML notation for an include relationship is a dashed arrow from the source (including) use case to the target (included) use case. The keyword «include» annotates the arrow. Figure 8.1 shows an example from an online stock brokerage system. Part of establishing a secure session is validating the user password. In addition, the stock brokerage system validates the password for each stock trade. Use cases secure session and make trade both include use case validate password.

A use case can also be inserted within a textual description with the notation include use-case-name. An included use case is inserted at a specific location within the behavior sequence of the larger use case, just as a subroutine is called from a specific location within another subroutine.