# 1

# 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 multiprocessing 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.

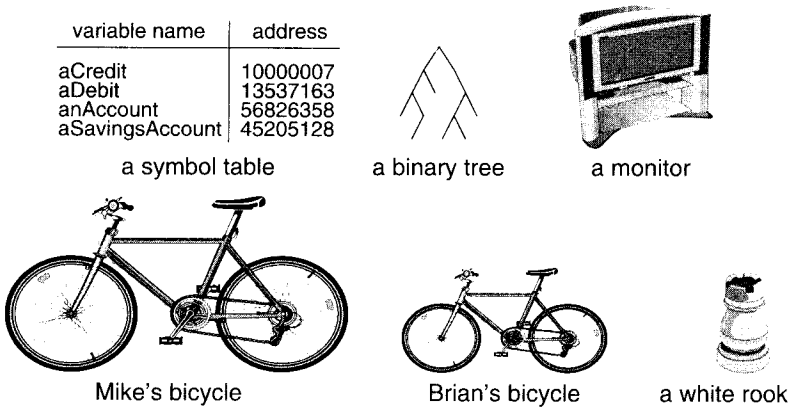


**Figure 1.1 Objects**. Objects lie at the heart of object-oriented technology.

***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

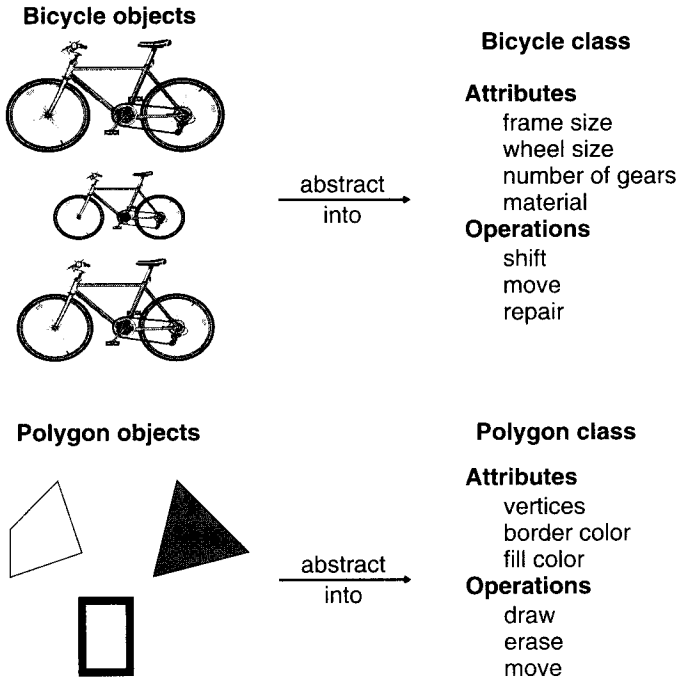


**Figure 1.2 Objects and classes.** Each class describes a possibly infinite set of individual objects.

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.

- **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 implementa-

tion 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).

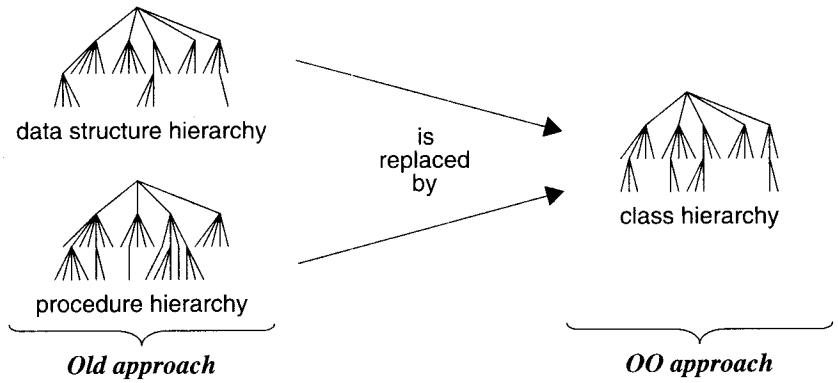


**Figure 1.3 OO vs. prior approach.** An OO approach has one unified hierarchy for both data and behavior.

### 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 con-

cepts, not fine details. We 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

[Brooks-95] Frederick P. Brooks, Jr. *The Mythical Man-Month, Anniversary Edition*. Boston: Addison-Wesley, 1995.

[Jacobson-99] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process*. Boston: Addison-Wesley, 1999.

[Love-93] Tom Love. *Object Lessons: Lessons Learned in Object-Oriented Development Practices*. New York: SIGS Books, 1993.

[Meyer-97] Bertrand Meyer. *Object-Oriented Software Construction, Second Edition*. Hertfordshire, England: Prentice Hall International, 1997.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

[Taylor-98] David A. Taylor. *Object Technology: A Manager's Guide, Second Edition*. Boston: Addison-Wesley, 1998.

## 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



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.

# 2

# 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.

- **Visualization**. Storyboards of movies, television shows, and advertisements let writers see how their ideas flow. They can modify awkward transitions, dangling ends, and unnecessary segments before detailed writing begins. Artists' sketches let them block out their ideas and make changes before committing them to oil or stone.
- **Reduction of complexity**. Perhaps the main reason for modeling, which incorporates all the previous reasons, is to deal with systems that are too complex to understand directly. The human mind can cope with only a limited amount of information at one time. Models reduce complexity by separating out a small number of important things to deal with at a time.

## 2.2 Abstraction

Abstraction is the selective examination of certain aspects of a problem. The goal of abstraction is to isolate those aspects that are important for some purpose and suppress those aspects that are unimportant. Abstraction must always be for some purpose, because the purpose determines what is, and is not, important. Many different abstractions of the same thing are possible, depending on the purpose for which they are made.

All abstractions are incomplete and inaccurate. Reality is a seamless web. Anything we say about it, any description of it, is an abridgement. All human words and language are abstractions—incomplete descriptions of the real world. This does not destroy their usefulness. The purpose of an abstraction is to limit the universe so we can understand. In building models, therefore, you must not search for absolute truth but for adequacy for some purpose. There is no single "correct" model of a situation, only adequate and inadequate ones.

A good model captures the crucial aspects of a problem and omits the others. Most computer languages, for example, are poor vehicles for modeling algorithms because they force the specification of implementation details that are irrelevant to the algorithm. A model that contains extraneous detail unnecessarily limits your choice of design decisions and diverts attention from the real issues.

## 2.3 The Three Models

We find it useful to model a system from three related but different viewpoints, each capturing important aspects of the system, but all required for a complete description. The *class model* represents the static, structural, "data" aspects of a system. The *state model* represents the temporal, behavioral, "control" aspects of a system. The *interaction model* represents the collaboration of individual objects, the "interaction" aspects of a system. A typical software procedure incorporates all three aspects: It uses data structures (class model), it sequences operations in time (state model), and it passes data and control among objects (interaction model). Each model contains references to entities in other models. For example, the class model attaches operations to classes, while the state and interaction models elaborate the operations.

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 ply, 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 unclog 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

h. Weight
i. Availability
j. Strength
k. Resistance to high temperatures
l. Resistance to stretching

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.

***Class diagrams*** provide a graphic notation for modeling classes and their relationships, thereby describing possible objects. Class diagrams are useful both for abstract modeling and for designing actual programs. They are concise, easy to understand, and work well in practice. We will use class diagrams throughout this book to represent the structure of applications.

We will also occasionally use object diagrams. An ***object diagram*** shows individual objects and their relationships. Object diagrams are helpful for documenting test cases and discussing examples. A class diagram corresponds to an infinite set of object diagrams.

Figure 3.1 shows a class (left) and instances (right) described by it. Objects *JoeSmith*, *MarySharp*, and an anonymous person are instances of class *Person*. The UML symbol for an object is a box with an object name followed by a colon and the class name. The object name and class name are both underlined. Our convention is to list the object name and class name in boldface.

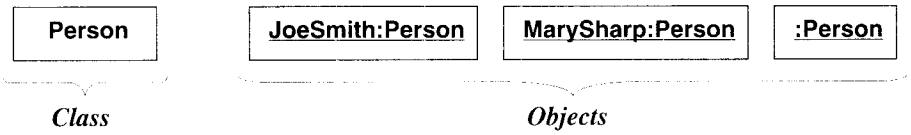


**Figure 3.1 A class and objects.** Objects and classes are the focus of class modeling.

The UML symbol for a class also is a box. Our convention is to list the class name in boldface, center the name in the box, and capitalize the first letter. We use singular nouns for the names of classes.

Note how we run together multiword names, such as *JoeSmith*, separating the words with intervening capital letters. This is the convention we use for referring to objects, classes, and other constructs. Alternative conventions would be to use intervening spaces (Joe Smith) or underscores (Joe_Smith). The mixed capitalization convention is popular in the OO literature but is not a UML requirement.

### 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*.

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.

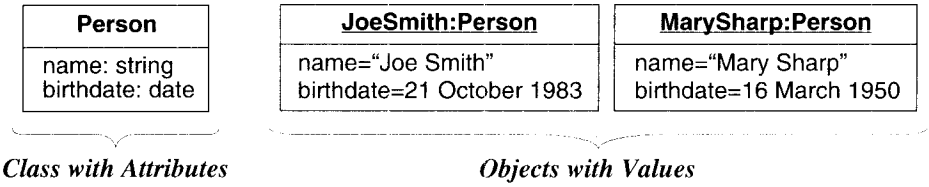


**Figure 3.2 Attributes and values.** Attributes elaborate classes.

The UML notation lists attributes in the second compartment of the class box. Optional details, such as type and default value, may follow each attribute. A colon precedes the type. An equal sign precedes the default value. Our convention is to show the attribute name in regular face, left align the name in the box, and use a lowercase letter for the first letter.

You may also include attribute values in the second compartment of object boxes. The notation is to list each attribute name followed by an equal sign and the value. We also left align attribute values and use regular type face.

Some implementation media require that an object have a unique identifier. These identifiers are implicit in a class model—you need not and should not list them explicitly. Figure 3.3 emphasizes the point. Most OO languages automatically generate identifiers with which to reference objects. You can also readily define them for databases. Identifiers are a computer artifact and have no intrinsic meaning.

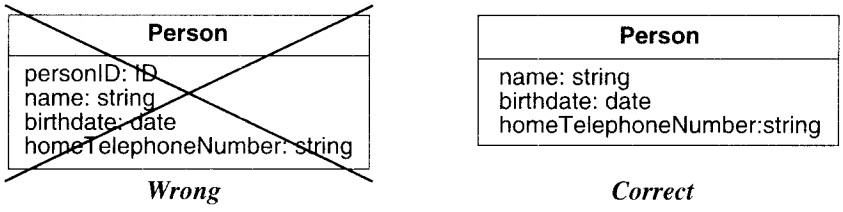


**Figure 3.3 Object identifiers.** Do not list object identifiers; they are implicit in models.

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.

### 3.1.5 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.

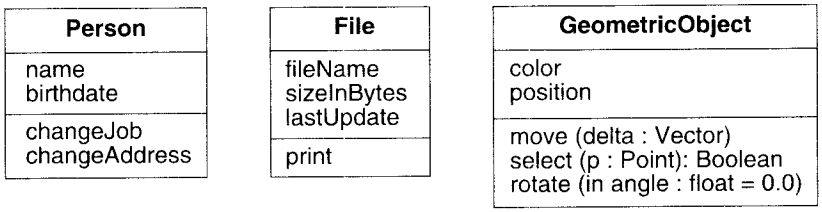


**Figure 3.4 Operations.** An operation is a function or procedure that may be applied to or by objects in a class.

### 3.1.6 Summary of Notation for Classes

Figure 3.5 summarizes the notation for classes. A box represents a class and may have as many as three compartments. The compartments contain, from top to bottom: class name, list of attributes, and list of operations. Optional details such as type and default value may follow each attribute name. Optional details such as argument list and result type may follow each operation name.

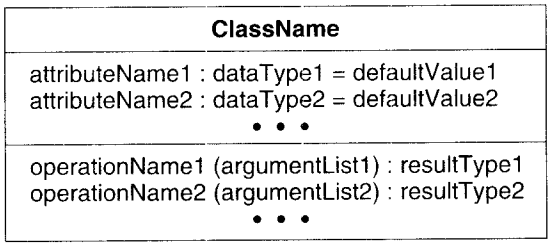


**Figure 3.5 Summary of modeling notation for classes.** A box represents a class and may have as many as three compartments.

Figure 3.6 shows that each argument may have a direction, name, type, and default value. The ***direction*** indicates whether an argument is an input (*in*), output (*out*), or an input argument that can be modified (*inout*). A colon precedes the type. An equal sign precedes the default value. The default value is used if no argument is supplied for the argument.

direction argumentName : type = defaultValue

**Figure 3.6 Notation for an argument of an operation.** The direction, type, and default value are optional. Direction may be *in*, *out*, or *inout*.

The attribute and operation compartments of class boxes are optional, and you may or may not show them. A missing attribute compartment means that attributes are unspecified. Similarly, a missing operation compartment means that operations are unspecified. In contrast, an empty compartment means that attributes (operations) are specified and that there are none.

## 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.

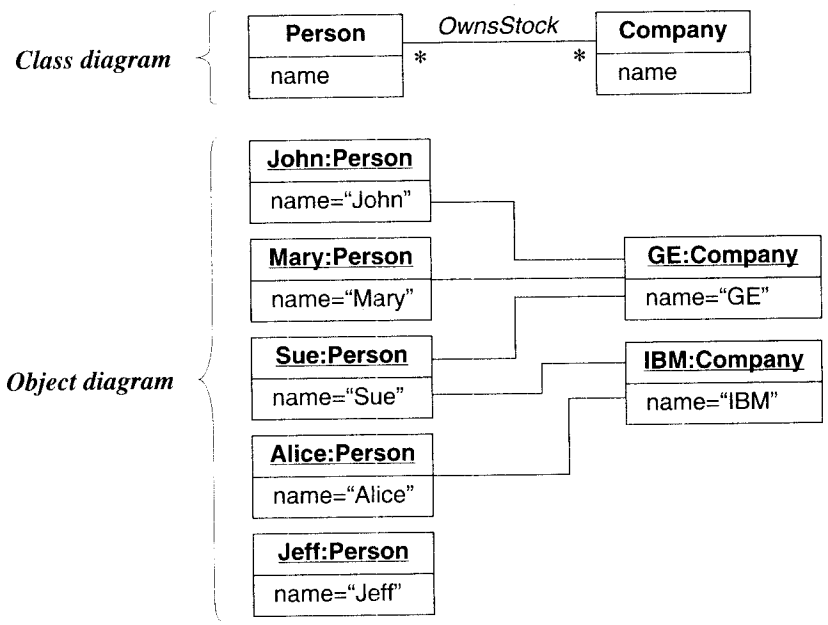


**Figure 3.7 Many-to-many association.** An association describes a set of potential links in the same way that a class describes a set of potential objects.

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 constrains 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.)

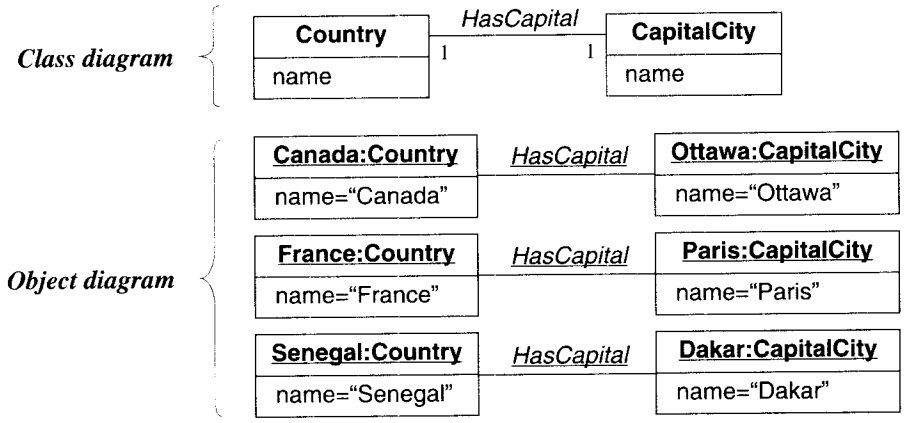


**Figure 3.8 One-to-one association.** Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class.

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.)


**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.

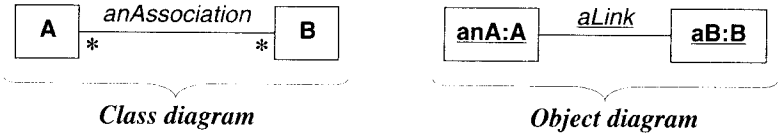


**Figure 3.10 Association vs. link.** A pair of objects can be instantiated at most once per association (except for bags and sequences).

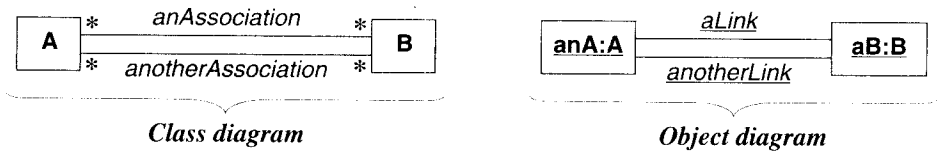


**Figure 3.11 Association vs. link.** You can use multiple associations to model multiple links between the same objects.

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.

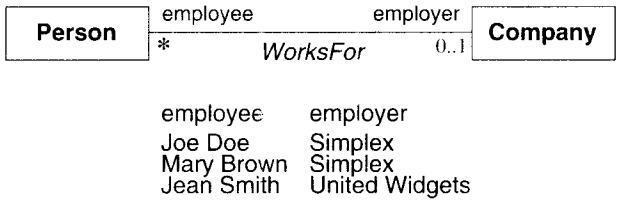


**Figure 3.12 Association end names.** Each end of an association can have a name.

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 *container* and *contents* 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.

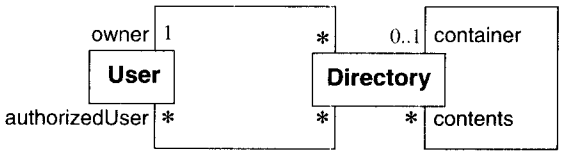


**Figure 3.13 Association end names.** Association end names are necessary for associations between two objects of the same class. They can also distinguish multiple associations between a pair of classes.

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.)

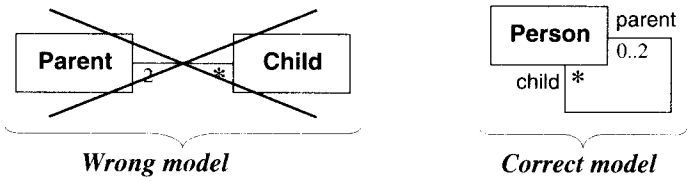


**Figure 3.14 Association end names.** Use association end names to model multiple references to the same class.

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.

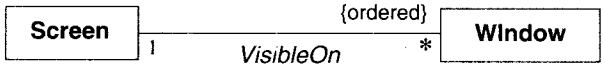


**Figure 3.15 Ordering the objects for an association end.** Ordering sometimes occurs for "many" multiplicity.

### 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.

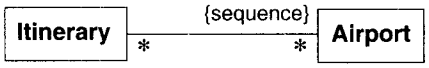


**Figure 3.16 An example of a sequence.** An itinerary may visit multiple airports, so you should use *{sequence}* and not *{ordered}*.

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, *accessPermission* is an attribute of *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.

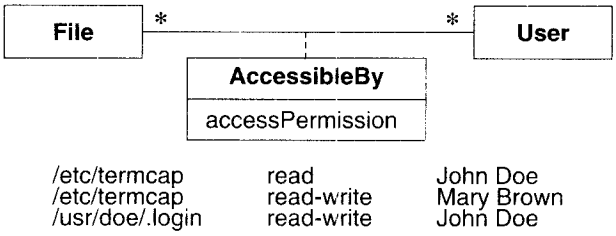


**Figure 3.17 An association class.** The links of an association can have attributes.

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.

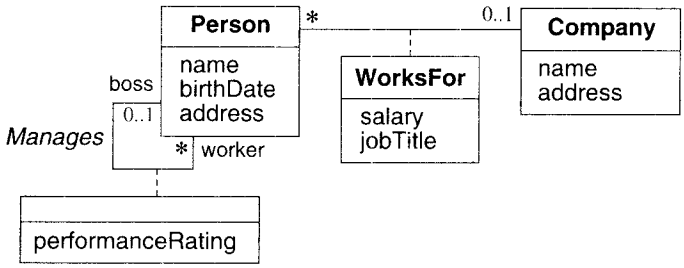


**Figure 3.18 Association classes.** Attributes may also occur for one-to-many and one-to-one associations.

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.

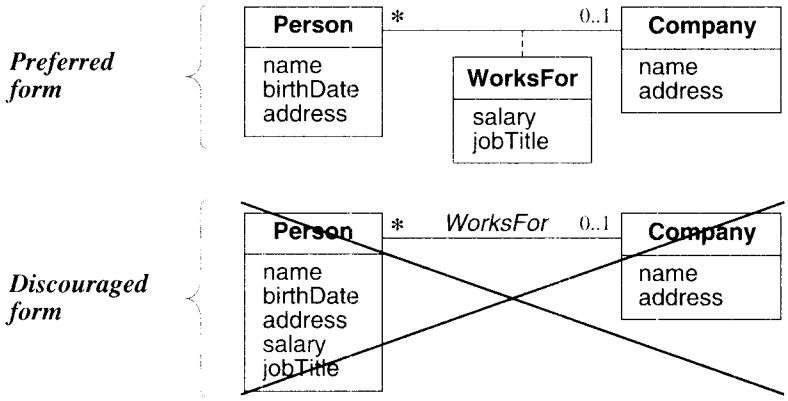


**Figure 3.19 Proper use of association classes.** Do not fold attributes of an association into a class.

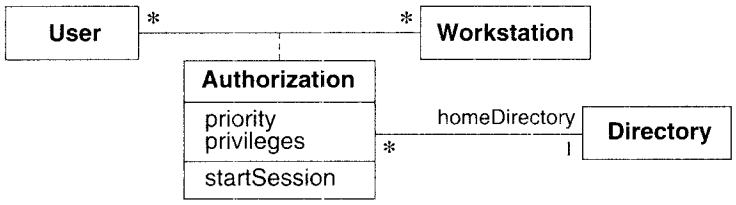


**Figure 3.20 An association class participating in an association.** Association classes let you specify identity and navigation paths precisely.

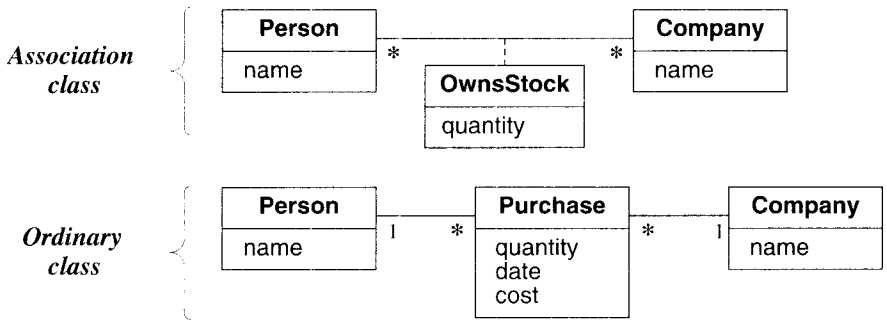


**Figure 3.21 Association class vs. ordinary class.** An association class is much different than an ordinary class.

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.

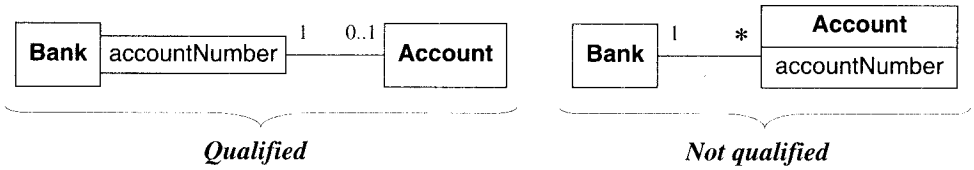


**Figure 3.22 Qualified association**. Qualification increases the precision of a model.

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 presuming this is true. If every stock had a single ticker symbol that was invariant across exchanges, we would make *tickerSymbol* an attribute of *Company*.)

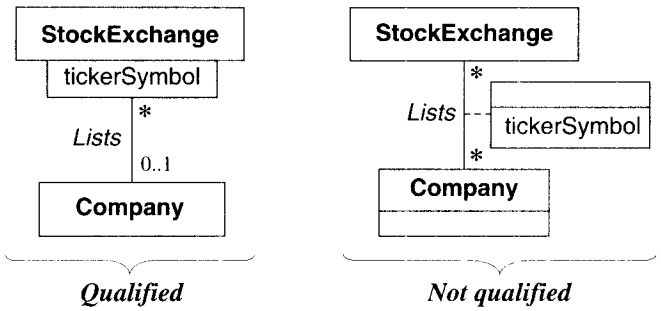


**Figure 3.23 Qualified association.** Qualification also facilitates traversal of class models.

# 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 superclasses.) 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.

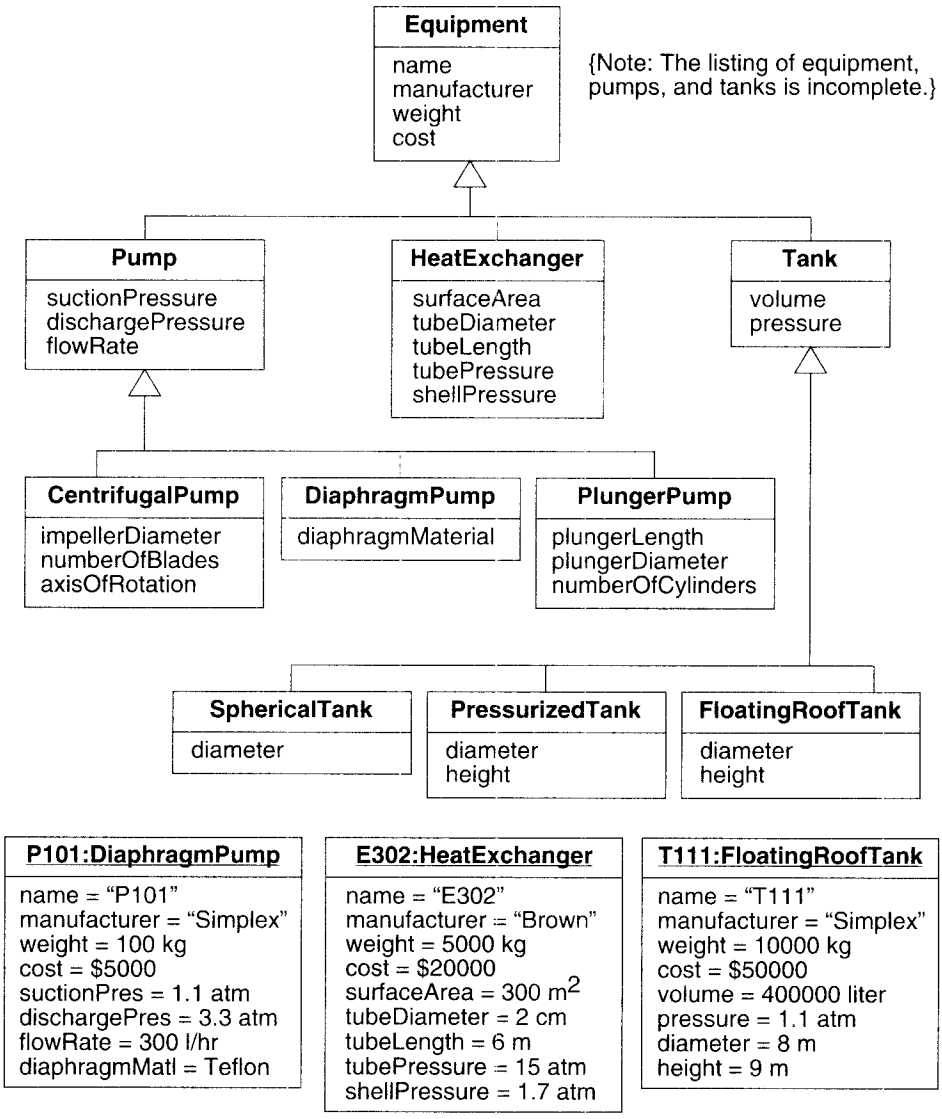


**Figure 3.24 A multilevel inheritance hierarchy with instances.** Generalization organizes classes by their similarities and differences, structuring the description of objects.

Generalization is transitive across an arbitrary number of levels. The terms ***ancestor*** and ***descendant*** refer to generalization of classes across multiple levels. An instance of a subclass is simultaneously an instance of all its ancestor classes. An instance includes a value for every attribute of every ancestor class. An instance can invoke any operation on any ancestor

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.

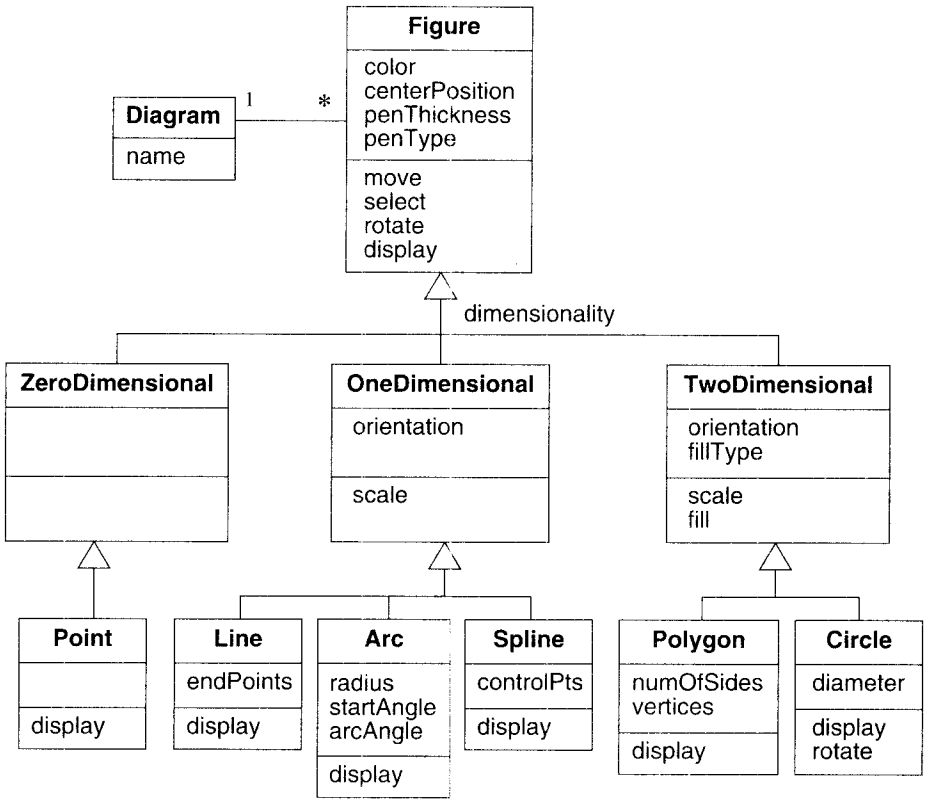


**Figure 3.25 Inheritance for graphic figures.** Each subclass inherits the attributes, operations, and associations of its superclasses.

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, cy1, cx2, 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 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-

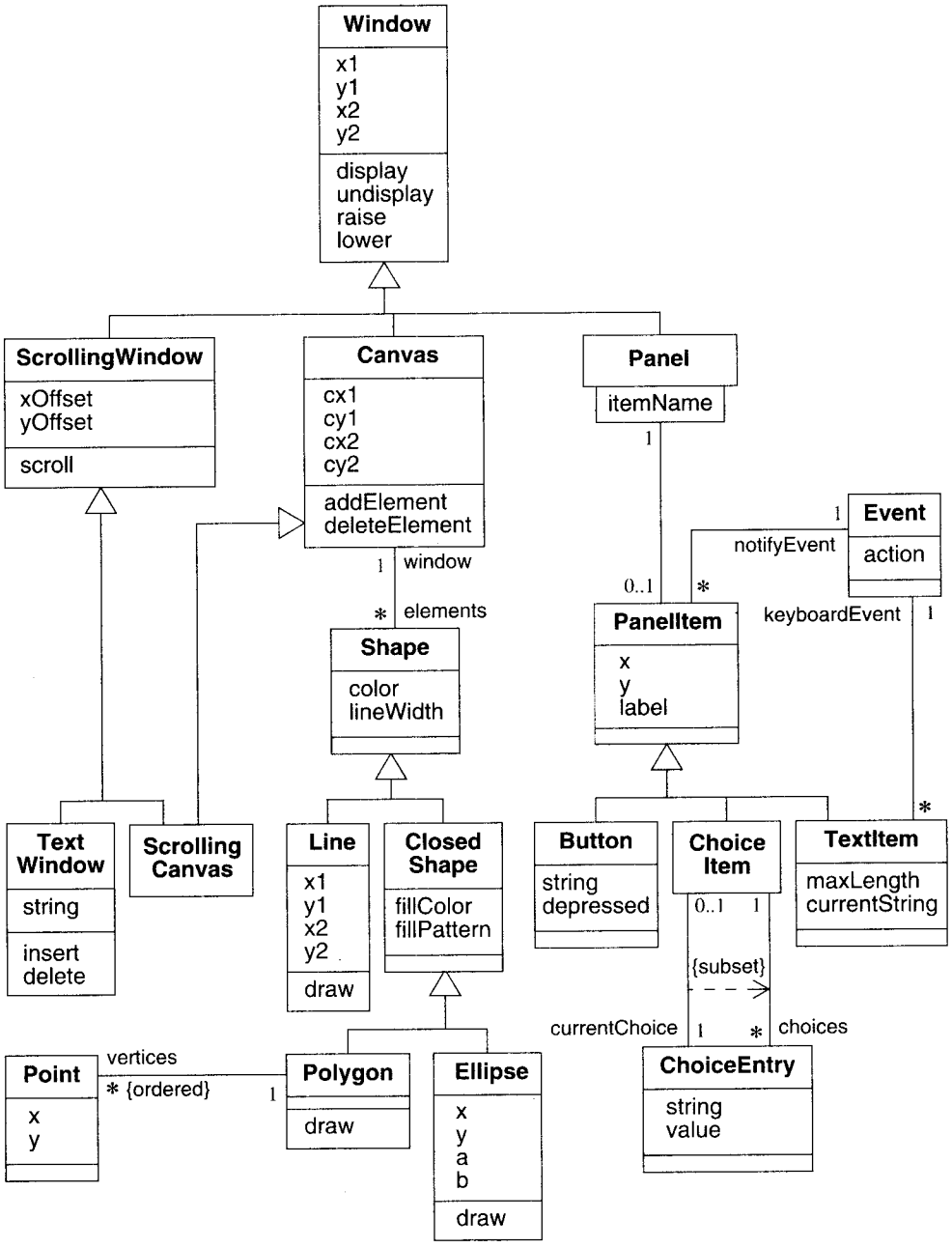


**Figure 3.26 Class model of a windowing system**

sociation 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?

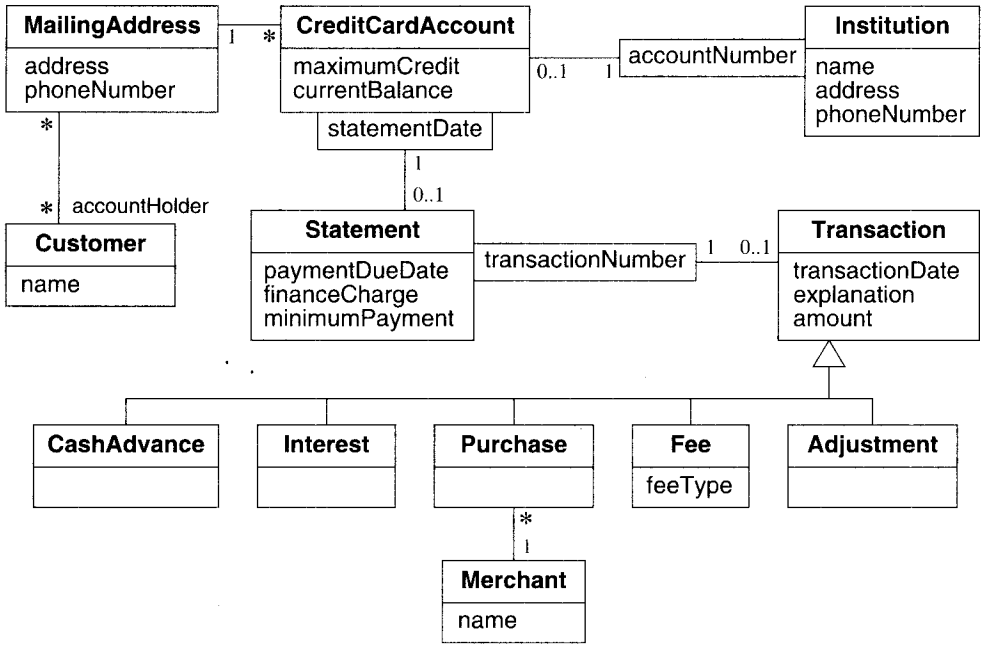


**Figure 3.27 Class model for managing credit card accounts**

- 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)*** [Warmer-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 *dot* 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[30 November 1999]* 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

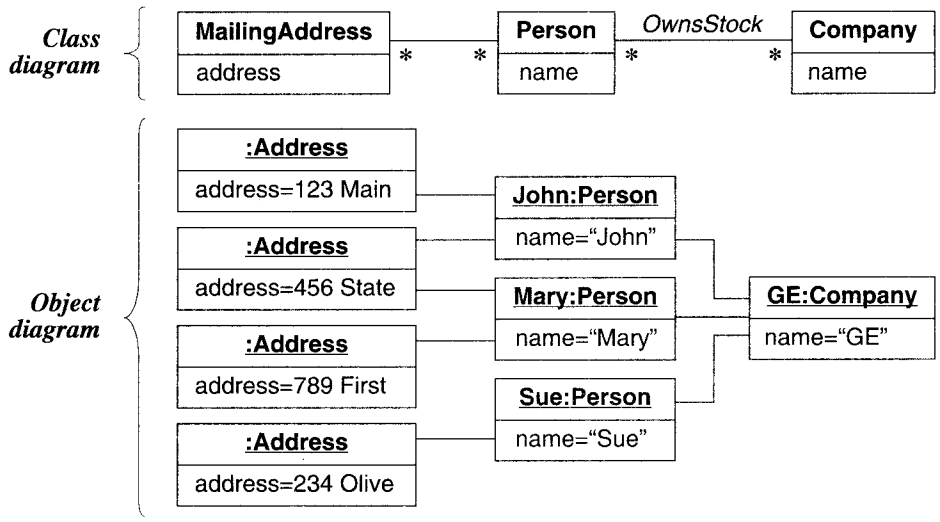


**Figure 3.28 A sample model and examples.** Traversal of multiple associations can yield a bag.

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

[Warmer-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?

```
aCreditCardAccount.Statement.Transaction->
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 *aStartDate* and *anEndDate*.

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

```
anInstitution.CreditCardAccount.Statement.Transaction->
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?

```
aMerchant.Purchase->
select(aStartDate <= transactionDate and
transactionDate <= anEndDate).Statement.
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?

```
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?

```
aCustomer.MailingAddress.CreditCardAccount.
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)

- **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.

## 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

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.

| | | | |
|---|---|---|---|
| ancestor | default value | link | polymorphism |
| association | descendant | method | qualified association |
| association class | direction | multiplicity | qualifier |
| association end | feature | navigation | sequence |
| attribute | generalization | object | signature |
| bag | generalization set name | object diagram | specialization |
| class | identity | operation | subclass |
| class diagram | inheritance | ordering | superclass |
| class model | instance | override | value |

**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. As 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 IDEF1X, 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.

[Khoshafian-86] defines the concept of object identity and its importance to programming languages and database systems.

[Warmer-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

[Blaha-98] Michael Blaha and William Premerlani. *Object-Oriented Modeling and Design for Database Applications*. Upper Saddle River, NJ: Prentice Hall, 1998.

[Booch-99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Boston: Addison-Wesley, 1999.

[Chen-76] P.P.S. Chen. The Entity-Relationship model—toward a unified view of data. *ACM Transactions on Database Systems 1,* 1 (March 1976), 9–36.

[Chonoles-03] Michael Jesse Chonoles and James A. Schardt. *UML2 for Dummies*. New York: Wiley, 2003.

[Fowler-00] Martin Fowler. *UML Distilled, Second Edition*. Boston: Addison-Wesley, 2000.

[Khoshafian-86] S.N. Khoshafian and G.P. Copeland. Object identity. *OOPLSA'86 as ACM SIGPLAN 21,* 11 (November 1986), 406–416.

[Larman-02] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Upper Saddle River, NJ: Prentice Hall, 2002.

[Loomis-87] Mary E.S. Loomis, Ashwin V. Shah, and James E. Rumbaugh. An object modeling technique for conceptual design. *European Conference on Object-Oriented Programming*, Paris, France, June 15–17, 1987, published as *Lecture Notes in Computer Science, 276,* Springer-Verlag, 192–202.

[Rayside-00] Derek Rayside and Gerard Campbell. An Aristotelian understanding of object-oriented programming. *OOPLSA'00 as ACM SIGPLAN 35,* 10 (October 2000), 337–353.

[Rumbaugh-87] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. *OOPSLA'87 as ACM SIGPLAN 22,* 12 (December1987), 466–481.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

[UML] www.uml.org

[Warmer-99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Boston: Addison-Wesley, 1999.

# Exercises

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

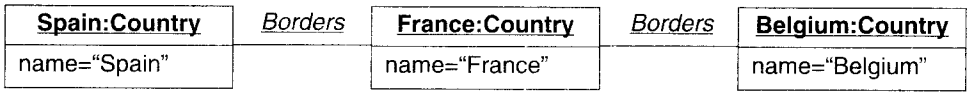


**Figure E3.1 Object diagram for a portion of Europe**

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.

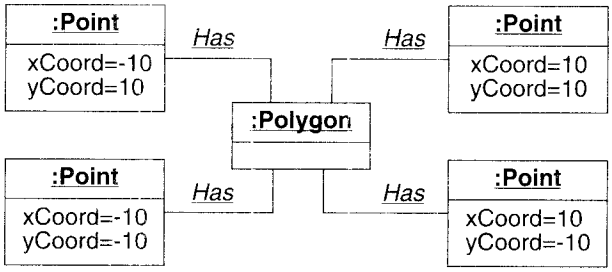


**Figure E3.2 Object diagram for a polygon that happens to be a square**

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.

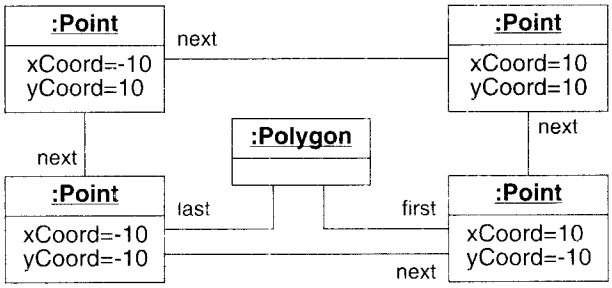


**Figure E3.3 Object diagram for a polygon that happens to be a square**

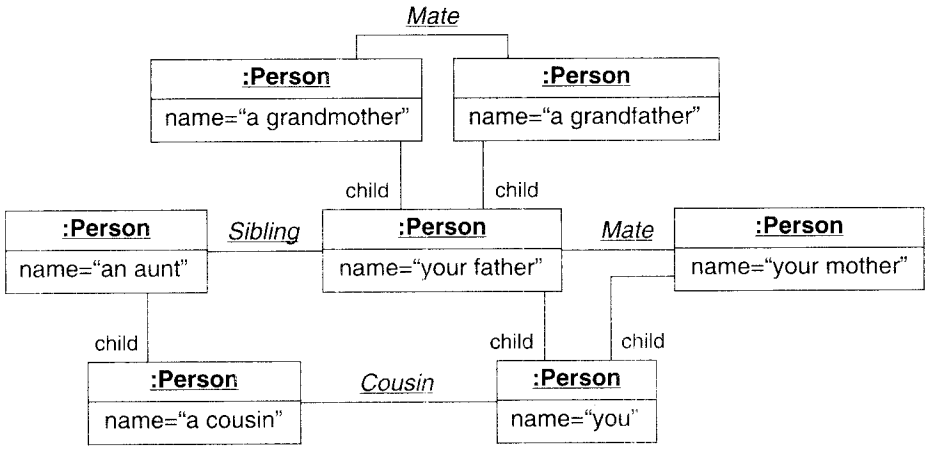


**Figure E3.4 Object diagram for part of your family tree**

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

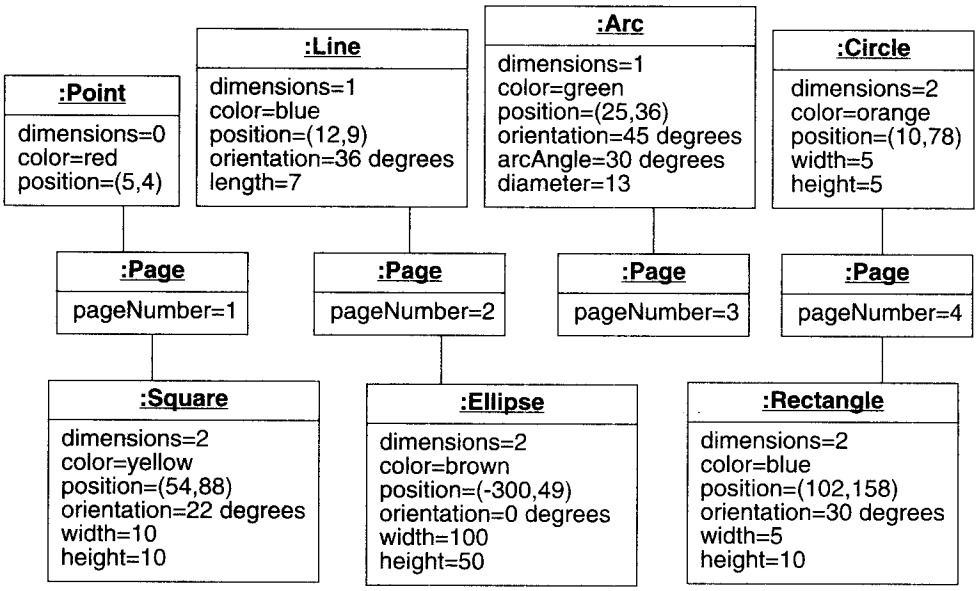


**Figure E3.5 Object diagram for a geometrical document**

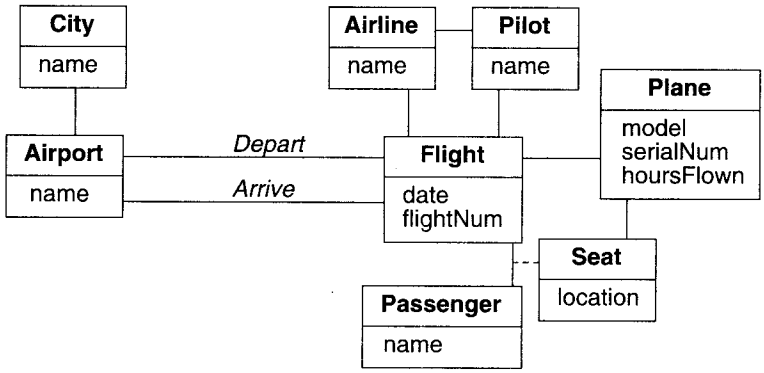


**Figure E3.6 Partially completed class model of an air transportation system**

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.

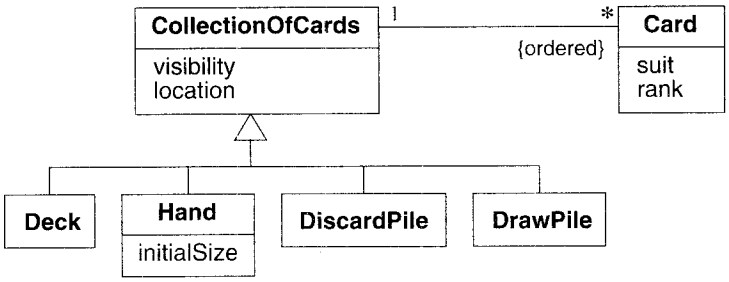


**Figure E3.7 Portion of a class diagram for a card-playing system**

3.16 (5) Figure E3.8 is a portion of a class diagram for a computer system for laying out a newspaper. The system handles newspaper pages which may contain, among other things, columns of text. The user may edit the width and length of a column of text, move it around on a page, or move it from one page to another. As shown, a column is displayed on exactly one page.

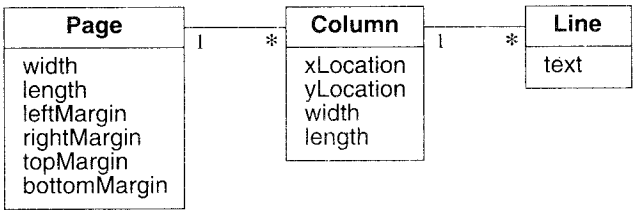


**Figure E3.8 Portion of a class diagram for a newspaper publishing system**

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.

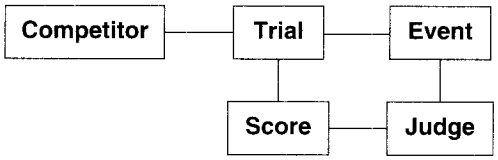


**Figure E3.9 Portion of a class diagram for an athletic-event scoring system**

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.

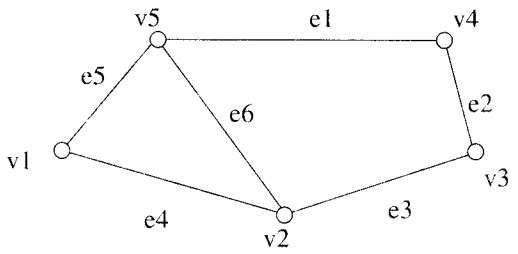


**Figure E3.10 Sample undirected graph**

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.

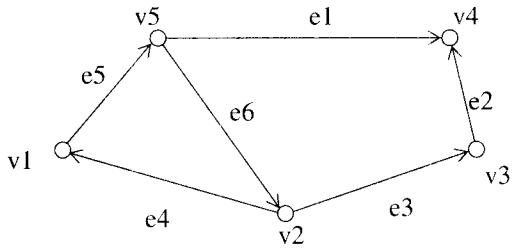


**Figure E3.11 Sample directed graph**

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.

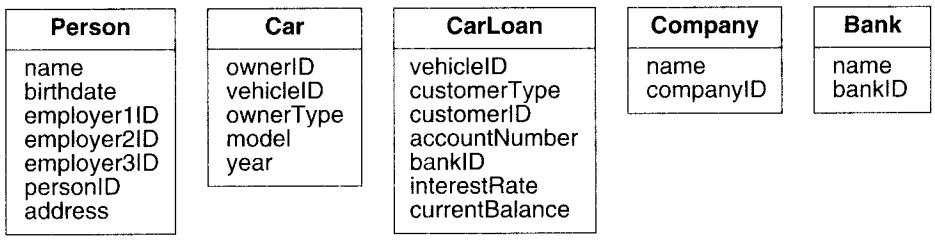


**Figure E3.12 Classes with some attributes that are references.**

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 shaft 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 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 listing 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-

sists 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(date)* 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.

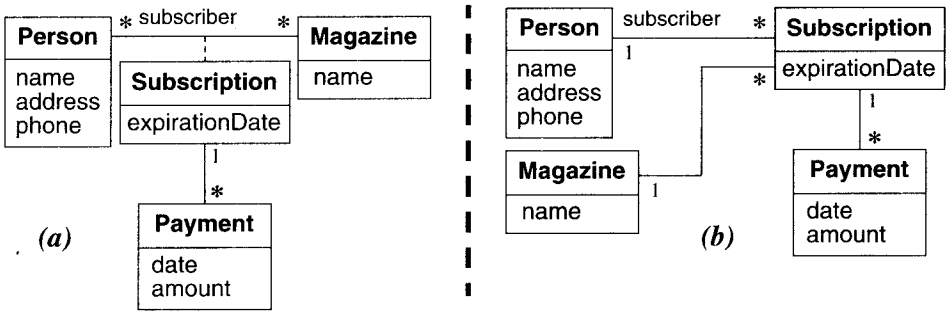


**Figure E3.13 Class diagram for magazine subscriptions**

# 4

# Advanced Class Modeling

Chapter 4 continues our discussion of class modeling concepts with a treatment of advanced topics. This chapter provides subtleties for improved modeling that you can skip upon a first reading of this book.

## 4.1 Advanced Object and Class Concepts

### 4.1.1 Enumerations

A data type is a description of values. Data types include numbers, strings, and enumerations. An ***enumeration*** is a data type that has a finite set of values. For example, the attribute *accessPermission* in Figure 3.17 is an enumeration with possible values that include *read* and *read-write*. Figure 3.25 also has some enumerations that Figure 4.1 illustrates. *Figure.penType* is an enumeration that includes *solid*, *dashed*, and *dotted*. *TwoDimensional.fillType* is an enumeration that includes *solid*, *grey*, *none*, *horizontal lines*, and *vertical lines*.

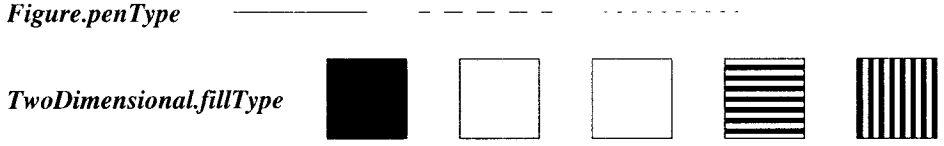


**Figure 4.1 Examples of enumerations.** Enumerations often occur and are important to users. Implementations must enforce the finite set of values.

When constructing a model, you should carefully note enumerations, because they often occur and are important to users. Enumerations are also significant for an implementation;

you may display the possible values with a pick list and you must restrict data to the legitimate values.

Do not use a generalization to capture the values of an enumerated attribute. An enumeration is merely a list of values; generalization is a means for structuring the description of objects. You should introduce generalization only when at least one subclass has significant attributes, operations, or associations that do not apply to the superclass. As Figure 4.2 shows, you should not introduce a generalization for *Card*, because most games do not differentiate the behavior of spades, clubs, hearts, and diamonds.

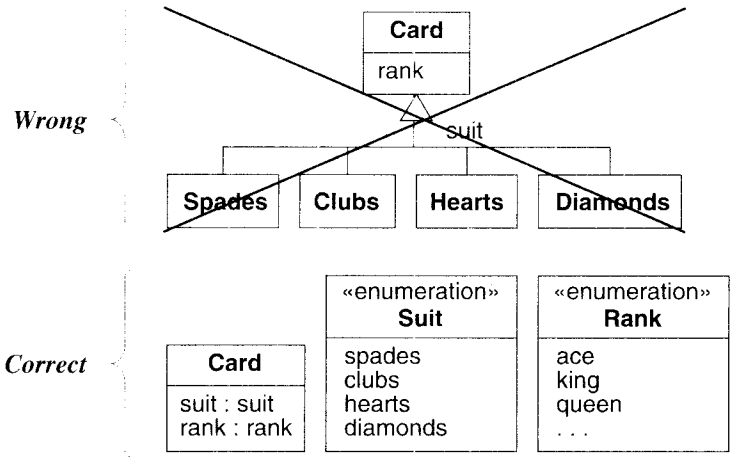


**Figure 4.2 Modeling enumerations.** Do not use a generalization to capture the values of an enumerated attribute.

In the UML an enumeration is a data type. You can declare an enumeration by listing the keyword *enumeration* in guillemets («») above the enumeration name in the top section of a box. The second section lists the enumeration values.

### 4.1.2 Multiplicity

Multiplicity is a constraint on the cardinality of a set. Chapter 3 explained multiplicity for associations. Multiplicity also applies to attributes.

It is often helpful to specify multiplicity for an attribute, especially for database applications. ***Multiplicity for an attribute*** specifies the number of possible values for each instantiation of an attribute. The most common specifications are a mandatory single value [1], an optional single value [0..1], and many [*]. Multiplicity specifies whether an attribute is mandatory or optional (in database terminology whether an attribute can be null). Multiplicity also indicates if an attribute is single valued or can be a collection. If not specified, an attribute is assumed to be a mandatory single value ([1]). In Figure 4.3 a person has one name, one or more addresses, zero or more phone numbers, and one birthdate.

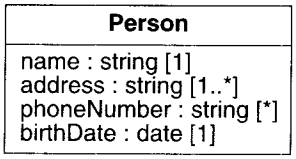


**Figure 4.3 Multiplicity for attributes.** You can specify whether an attribute is single or multivalued, mandatory or optional.

### 4.1.3 Scope

Chapter 3 presented features for individual objects. This is the default usage, but there can also be features for an entire class. The ***scope*** indicates if a feature applies to an object or a class. An underline distinguishes features with class scope (static) from those with object scope. Our convention is to list attributes and operations with class scope at the top of the attribute and operation boxes, respectively.

It is acceptable to use an attribute with class scope to hold the ***extent*** of a class (the set of objects for a class)—this is common with OO databases. Otherwise, you should avoid attributes with class scope because they can lead to an inferior model. It is better to model groups explicitly and assign attributes to them. For example, the upper model in Figure 4.4 shows a simple model of phone mail. Each message has an owner mailbox, date recorded, time recorded, priority, message contents, and a flag indicating if it has been received. A message may have a mailbox as the source or it may be from an external call. Each mailbox has a phone number, password, and recorded greeting. For the *PhoneMessage* class we can store the maximum duration for a message and the maximum days a message will be retained. For the *PhoneMailbox* class we can store the maximum number of messages that can be stored.

The upper model is inferior, however, because the maximum duration, maximum days retained, and maximum message count have a single value for the entire phone mail system. In the lower model these limits can vary for different kinds of users, yielding a more flexible and extensible phone mail system.

In contrast to attributes, it is acceptable to define operations of class scope. The most common use of class-scoped operations is to create new instances of a class. Sometimes it is convenient to define class-scoped operations to provide summary data. You should be careful with the use of class-scoped operations for distributed applications.

### 4.1.4 Visibility

***Visibility*** refers to the ability of a method to reference a feature from another class and has the possible values of *public*, *protected*, *private*, and *package*. The precise meaning depends on the programming language. (See Chapter 18 for details.) Any method can freely access ***public*** features. Only methods of the containing class and its descendants via inheritance can access ***protected*** features. (Protected features also have package accessibility in Java.) Only methods of the containing class can access ***private*** features. Methods of classes defined in the same package as the target class can access ***package*** features.

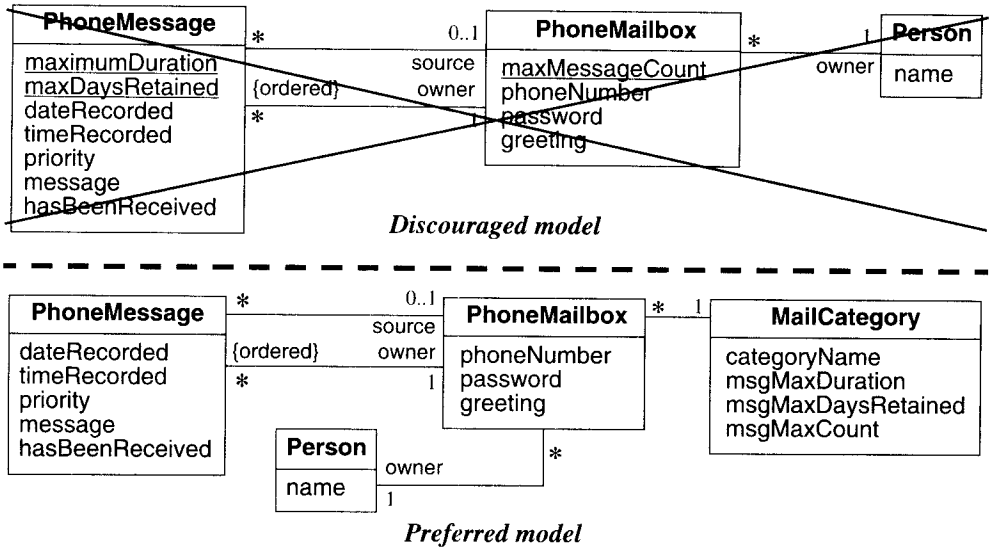


**Figure 4.4 Attribute scope.** Instead of assigning attributes to classes, model groups explicitly.

The UML denotes visibility with a prefix. The character "+" precedes public features. The character "#" precedes protected features. The character "-" precedes private features. And the character "~" precedes package features. The lack of a prefix reveals no information about visibility.

There are several issues to consider when choosing visibility.

- **Comprehension**. You must understand all public features to understand the capabilities of a class. In contrast, you can ignore private, protected, and package features—they are merely an implementation convenience.
- **Extensibility**. Many classes can depend on public methods, so it can be highly disruptive to change their signature (number of arguments, types of arguments, type of return value). Since fewer classes depend on private, protected, and package methods, there is more latitude to change them.
- **Context**. Private, protected, and package methods may rely on preconditions or state information created by other methods in the class. Applied out of context, a private method may calculate incorrect results or cause the object to fail.

## 4.2 Association Ends

As the name implies, an ***association end*** is an end of an association. A binary association has two ends, a ternary association (Section 4.3) has three ends, and so forth. Chapter 3 discussed the following properties.

- **Association end name**. An association end may have a name. The names disambiguate multiple references to a class and facilitate navigation. Meaningful names often arise, and it is useful to place the names within the proper context.
- **Multiplicity**. You can specify multiplicity for each association end. The most common multiplicities are "1" (exactly one), "0..1" (at most one), and "*" ("many"—zero or more).
- **Ordering**. The objects for a "many" association end are usually just a set. However, sometimes the objects have an explicit order.
- **Bags and sequences**. The objects for a "many" association end can also be a bag or sequence.
- **Qualification**. One or more qualifier attributes can disambiguate the objects for a "many" association end.

Association ends have some additional properties.

- **Aggregation**. The association end may be an aggregate or constituent part (Section 4.4). Only a binary association can be an aggregation; one association end must be an aggregate and the other must be a constituent.
- **Changeability**. This property specifies the update status of an association end. The possibilities are *changeable* (can be updated) and *readonly* (can only be initialized).
- **Navigability**. Conceptually, an association may be traversed in either direction. However, an implementation may support only one direction. The UML shows navigability with an arrowhead on the association end attached to the target class. Arrowheads may be attached to zero, one, or both ends of an association.
- **Visibility**. Similar to attributes and operations (Section 4.1.4), association ends may be *public*, *protected*, *private*, or *package*.

## 4.3 N-ary Associations

Chapter 3 presented binary associations (associations between two classes). However, you may occasionally encounter ***n-ary associations*** (associations among three or more classes.) You should try to avoid n-ary associations—most of them can be decomposed into binary associations, with possible qualifiers and attributes. Figure 4.5 shows an association that at first glance might seem to be an n-ary but can readily be restated as binary associations.

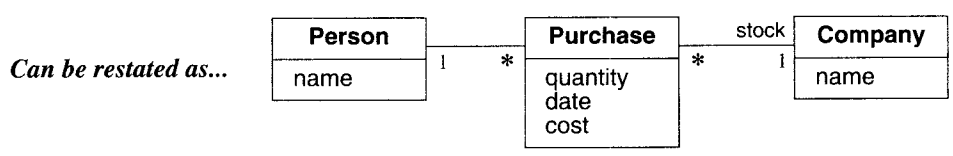


**Figure 4.5 Restating an n-ary association.** You can decompose most n-ary associations into binary associations.

Figure 4.6 shows a genuine n-ary (ternary) association: Programmers use computer languages on projects. This n-ary association is an atomic unit and cannot be subdivided into binary associations without losing information. A programmer may know a language and work on a project, but might not use the language on the project. The UML symbol for n-ary associations is a diamond with lines connecting to related classes. If the association has a name, it is written in italics next to the diamond.

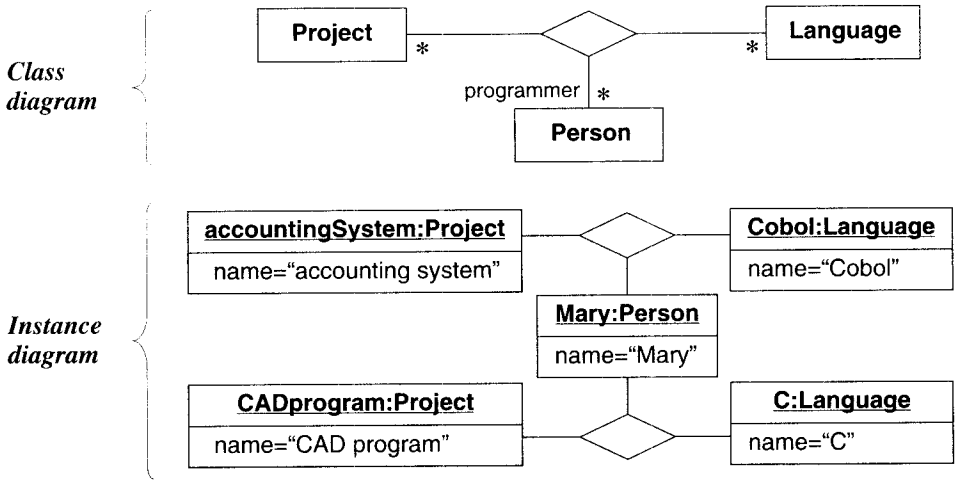


**Figure 4.6 Ternary association and links.** An n-ary association can have association end names, just like a binary association.

As Figure 4.6 illustrates, an n-ary association can have a name for each end just like a binary association. End names are necessary if a class participates in an n-ary association more than once. You cannot traverse n-ary associations from one end to another as with binary associations, so end names do not represent pseudo attributes of the participating classes. The OCL [Warmer-99] does not define notation for traversing n-ary associations.

Figure 4.7 shows another ternary association: A professor teaches a listed course during a semester. The resulting delivered course has a room number and any number of textbooks.

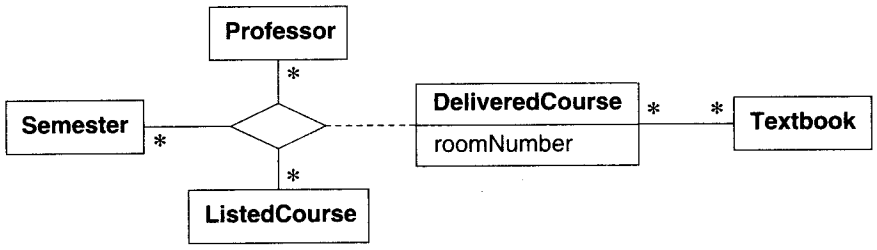


**Figure 4.7 Another ternary association.** N-ary associations are full-fledged associations and can have association classes.

The typical programming language cannot express n-ary associations. Thus if you are programming, you will need to promote n-ary associations to classes as Figure 4.8 does for *DeliveredClass*. Be aware that you change the meaning of a model, when you promote an n-ary association to a class. An n-ary association enforces that there is at most one link for each combination—for each combination of *Professor*, *Semester*, and *ListedCourse* in Figure 4.7 there is one *DeliveredCourse*. In contrast a promoted class permits any number of links—for each combination of *Professor*, *Semester*, and *ListedCourse* in Figure 4.8 there can be many *DeliveredCourses*. If you were implementing Figure 4.8, special application code would have to enforce the uniqueness of *Professor* + *Semester* + *ListedCourse*.

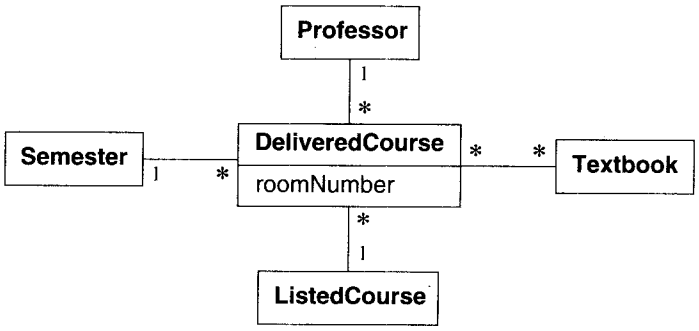


**Figure 4.8 Promoting an n-ary association.** Programming languages cannot express n-ary associations, so you must promote them to classes.

## 4.4 Aggregation

***Aggregation*** is a strong form of association in which an aggregate object is *made of* constituent parts. Constituents are *part of* the aggregate. The aggregate is semantically an extended object that is treated as a unit in many operations, although physically it is made of several lesser objects.

We define an aggregation as relating an assembly class to *one* constituent part class. An assembly with many kinds of constituent parts corresponds to many aggregations. For example, a *LawnMower* consists of a *Blade*, an *Engine*, many *Wheels*, and a *Deck*. *LawnMower* is the assembly and the other parts are constituents. *LawnMower* to *Blade* is one aggregation, *LawnMower* to *Engine* is another aggregation, and so on. We define each individual pairing as an aggregation so that we can specify the multiplicity of each constituent part within the assembly. This definition emphasizes that aggregation is a special form of binary association.

The most significant property of aggregation is ***transitivity***—that is, if *A* is part of *B* and *B* is part of *C*, then *A* is part of *C*. Aggregation is also ***antisymmetric***—that is, if *A* is part of *B*, then *B* is not part of *A*. Many aggregate operations imply transitive closure* and operate on both direct and indirect parts.

### 4.4.1 Aggregation Versus Association

Aggregation is a special form of association, not an independent concept. Aggregation adds semantic connotations. If two objects are tightly bound by a part-whole relationship, it is an aggregation. If the two objects are usually considered as independent, even though they may often be linked, it is an association. Some tests include:

- Would you use the phrase *part of*?
- Do some operations on the whole automatically apply to its parts?
- Do some attribute values propagate from the whole to all or some parts?
- Is there an intrinsic asymmetry to the association, where one class is subordinate to the other?

Aggregations include bill-of-materials, part explosions, and expansions of an object into constituent parts. Aggregation is drawn like association, except a small diamond indicates the assembly end. In Figure 4.9 a lawn mower consists of one blade, one engine, many wheels, and one deck. The manufacturing process is flexible and largely combines standard parts, so blades, engines, wheels, and decks pertain to multiple lawn mower designs.

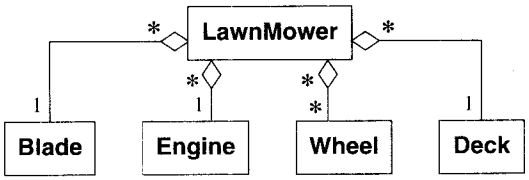


**Figure 4.9 Aggregation.** Aggregation is a kind of association in which an aggregate object is made of constituent parts.

The decision to use aggregation is a matter of judgment and can be arbitrary. Often it is not obvious if an association should be modeled as an aggregation. To a large extent this kind of uncertainty is typical of modeling; modeling requires seasoned judgment and there are few hard and fast rules. Our experience has been that if you exercise careful judgment and are consistent, the imprecise distinction between aggregation and ordinary association does not cause problems in practice.

### 4.4.2 Aggregation Versus Composition

The UML has two forms of part-whole relationships: a general form called *aggregation* and a more restrictive form called *composition*.

---

* Transitive closure is a term from graph theory. If $E$ denotes an edge and $N$ denotes a node and $S$ is the set of all pairs of nodes connected by an edge, then $S^+$ (the transitive closure of $S$) is the set of all pairs of nodes directly or indirectly connected by a sequence of edges. Thus $S^+$ includes all nodes that are directly connected, nodes connected by two edges, nodes connected by three edges, and so forth.

***Composition*** is a form of aggregation with two additional constraints. A constituent part can belong to at most one assembly. Furthermore, once a constituent part has been assigned an assembly, it has a coincident lifetime with the assembly. Thus composition implies ownership of the parts by the whole. This can be convenient for programming: Deletion of an assembly object triggers deletion of all constituent objects via composition. The notation for composition is a small *solid* diamond next to the assembly class (vs. a small *hollow* diamond for the general form of aggregation).

In Figure 4.10 a company consists of divisions, which in turn consist of departments; a company is indirectly a composition of departments. A company is not a composition of its employees, since company and person are independent objects of equal stature.

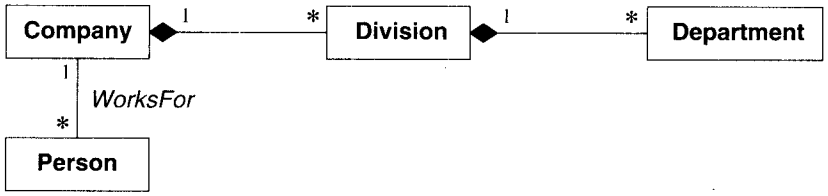


**Figure 4.10 Composition.** With composition a constituent part belongs to at most one assembly and has a coincident lifetime with the assembly.

### 4.4.3 Propagation of Operations

***Propagation*** (also called ***triggering***) is the automatic application of an operation to a network of objects when the operation is applied to some starting object [Rumbaugh-88].† For example, moving an aggregate moves its parts; the move operation propagates to the parts. Propagation of operations to parts is often a good indicator of aggregation.

Figure 4.11 shows an example of propagation. A person owns multiple documents. Each document consists of paragraphs that, in turn, consist of characters. The copy operation propagates from documents to paragraphs to characters. Copying a paragraph copies all the characters in it. The operation does not propagate in the reverse direction; a paragraph can be copied without copying the whole document. Similarly, copying a document copies the owner link but does not spawn a copy of the person who is owner.

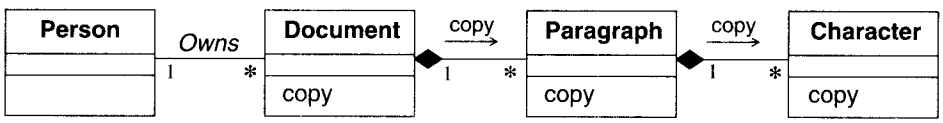


**Figure 4.11 Propagation.** You can propagate operations across aggregations and compositions.

† The term *association* as used in this book is synonymous with the term *relation* used in [Rumbaugh-88].

Most other approaches present an all-or-nothing option: copy an entire network with deep copy, or copy the starting object and none of the related objects with shallow copy. The concept of propagation of operations provides a concise and powerful way for specifying a continuum of behavior. You can think of an operation as starting at some initial object and flowing from object to object through links according to propagation rules. Propagation is possible for other operations including save/restore, destroy, print, lock, and display.

You can indicate propagation on class models with a small arrow indicating the direction and operation name next to the affected association. The notation binds propagation behavior to an association (or aggregation), direction, and operation. Note that this notation is not part of the UML and is a special notation.

## 4.5 Abstract Classes

An ***abstract class*** is a class that has no direct instances but whose descendant classes have direct instances. A ***concrete class*** is a class that is instantiable; that is, it can have direct instances. A concrete class may have abstract subclasses (but they, in turn, must have concrete descendants). Only concrete classes may be leaf classes in an inheritance tree.

All the occupations shown in Figure 4.12 are concrete classes. *Butcher, Baker,* and *CandlestickMaker* are concrete classes because they have direct instances. *Worker* also is a concrete class because some occupations may not be specified.

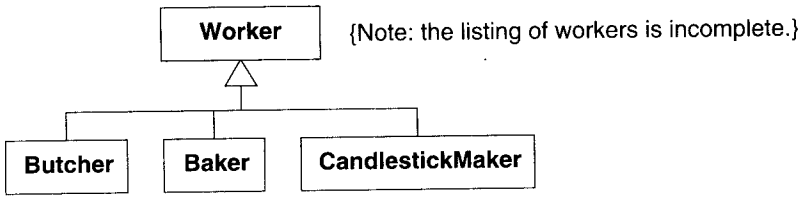


**Figure 4.12 Concrete classes.** A concrete class is instantiable; that is, it can have direct instances.

Class *Employee* in Figure 4.13 is an example of an abstract class. All employees must be either full-time or part-time. *FullTimeEmployee* and *PartTimeEmployee* are concrete classes because they can be directly instantiated. In the UML notation an abstract class name is listed in an italic font. Or you may place the keyword *{abstract}* below or after the name.

You can use abstract classes to define methods that can be inherited by subclasses. Alternatively, an abstract class can define the signature for an operation without supplying a corresponding method. We call this an ***abstract operation***. (Recall that an operation specifies the form of a function or procedure; a method is the actual implementation.) An abstract operation defines the signature of an operation for which each concrete subclass must provide its own implementation. A concrete class may not contain abstract operations, because objects of the concrete class would have undefined operations.

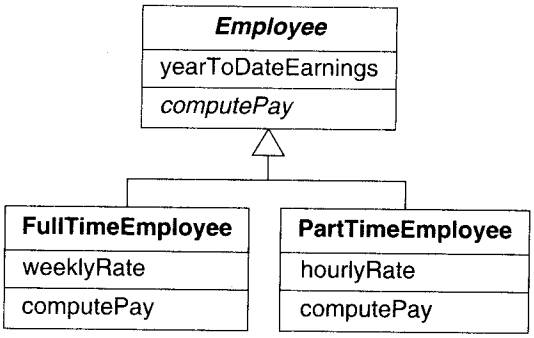


**Figure 4.13 Abstract class and abstract operation.** An abstract class is a class that has no direct instances

Figure 4.13 shows an abstract operation. An abstract operation is designated by italics or the keyword *{abstract}*. *ComputePay* is an abstract operation of class *Employee*; its signature but not its implementation is defined. Each subclass must supply a method for this operation.

Note that the abstract nature of a class is always provisional, depending on the point of view. You can always refine a concrete class into subclasses, making it abstract. Conversely, an abstract class may become concrete in an application in which the difference among its subclasses is unimportant.

As a matter of style, it is a good idea to avoid concrete superclasses. Then, abstract and concrete classes are readily apparent at a glance; all superclasses are abstract and all leaf subclasses are concrete. Furthermore, you will avoid awkward situations where a concrete superclass must both specify the signature of an operation for descendant classes and also provide an implementation for its concrete instances. You can always eliminate concrete superclasses by introducing an *Other* subclass, as Figure 4.14 shows.

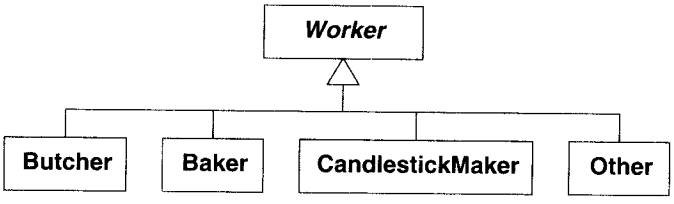


**Figure 4.14 Avoiding concrete superclasses.** You can always eliminate concrete superclasses by introducing an *Other* subclass.

## 4.6 Multiple Inheritance

***Multiple inheritance*** permits a class to have more than one superclass and to inherit features from all parents. Then you can mix information from two or more sources. This is a more

complicated form of generalization than single inheritance, which restricts the class hierarchy to a tree. The advantage of multiple inheritance is greater power in specifying classes and an increased opportunity for reuse. The disadvantage is a loss of conceptual and implementation simplicity.

The term *multiple inheritance* is used somewhat imprecisely to mean either the conceptual relationship between classes or the language mechanism that implements that relationship. Whenever possible, we try to distinguish between *generalization* (the conceptual relationship) and *inheritance* (the language mechanism), but the term "multiple inheritance" is more widely used than the term "multiple generalization."

### 4.6.1 Kinds of Multiple Inheritance

The most common form of multiple inheritance is from sets of disjoint classes. Each subclass inherits from one class in each set. In Figure 4.15 *FullTimeIndividualContributor* is both *FullTimeEmployee* and *IndividualContributor* and combines their features. *FullTimeEmployee* and *PartTimeEmployee* are disjoint; each employee must belong to exactly one of these. Similarly, *Manager* and *IndividualContributor* are also disjoint and each employee must be one or the other. The model does not show it, but we could define three additional combinations: *FullTimeManager*, *PartTimeIndividualContributor*, and *PartTimeManager*. The appropriate combinations depend on the needs of an application.

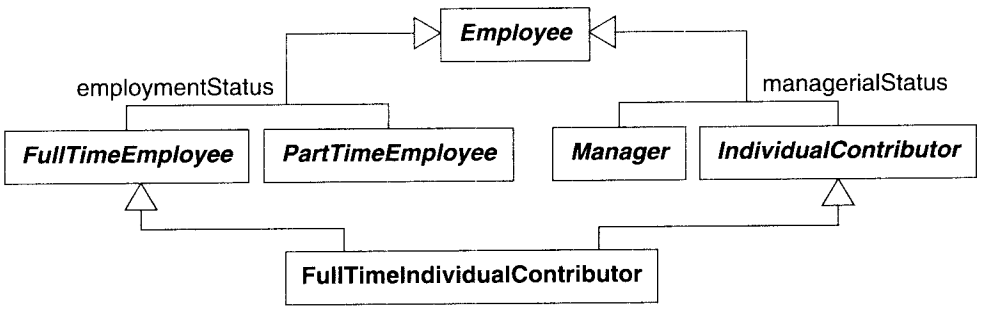


**Figure 4.15 Multiple inheritance from disjoint classes.** This is the most common form of multiple inheritance.

Each generalization should cover a single aspect. You should use multiple generalizations if a class can be refined on several distinct and independent aspects. In Figure 4.15, class *Employee* independently specializes on employment status and managerial status. Consequently the model has two separate generalization sets.

A subclass inherits a feature from the same ancestor class found along more than one path only once; it is the same feature. For example, in Figure 4.15 *FullTimeIndividualContributor* inherits *Employee* features along two paths, via *employmentStatus* and *managerialStatus*. However, each *FullTimeIndividualContributor* has only a single copy of *Employee* features.

Conflicts among parallel definitions create ambiguities that implementations must resolve. In practice, you should avoid such conflicts in models or explicitly resolve them, even if a particular language provides a priority rule for resolving conflicts. For example, suppose that *FullTimeEmployee* and *IndividualContributor* both have an attribute called *name*. *FullTimeEmployee.name* could refer to the person's full name while *IndividualContributor.name* might refer to the person's title. In principle, there is no obvious way to resolve such clashes. The best solution is to try to avoid them by restating the attributes as *FullTimeEmployee.personName* and *IndividualContributor.title*.

Multiple inheritance can also occur with overlapping classes. In Figure 4.16, *AmphibiousVehicle* is both *LandVehicle* and *WaterVehicle*. *LandVehicle* and *WaterVehicle* overlap, because some vehicles travel on both land and water. The UML uses a constraint (see Section 4.9) to indicate an overlapping generalization set; the notation is a dotted line cutting across the affected generalizations with keywords in braces. In this example, *overlapping* means that an individual vehicle may belong to more than one of the subclasses. *Incomplete* means that all possible subclasses of vehicle have not been explicitly named.

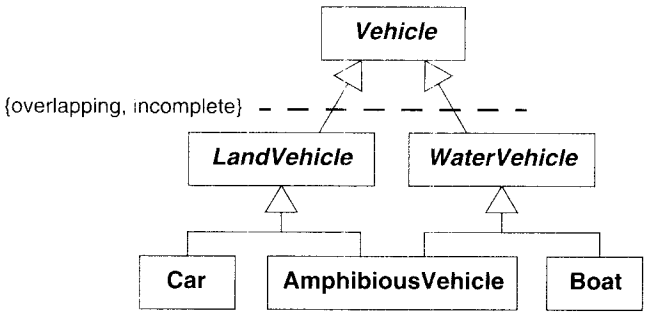


**Figure 4.16 Multiple inheritance from overlapping classes.** This form of multiple inheritance occurs less often than with disjoint classes.

### 4.6.2 Multiple Classification

An instance of a class is inherently an instance of all ancestors of the class. For example, an instructor could be both faculty and student. But what about a Harvard Professor taking classes at MIT? There is no class to describe the combination (it would be artificial to make one). This is an example of multiple classification, in which one instance happens to participate in two overlapping classes.

The UML permits multiple classification, but most OO languages handle it poorly. As Figure 4.17 shows, the best approach using conventional languages is to treat *Person* as an object composed of multiple *UniversityMember* objects. This workaround replaces inheritance with delegation (discussed in the next section). This is not totally satisfactory, because there is a loss of identity between the separate roles, but the alternatives involve radical changes in many programming languages [McAllester-86].

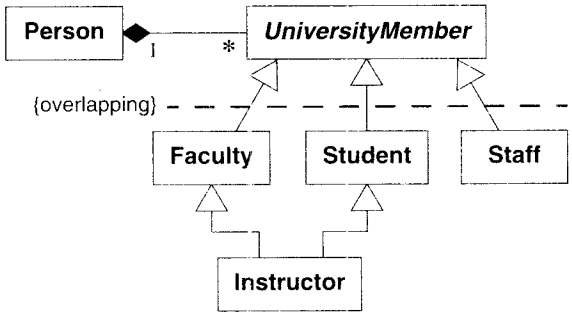


**Figure 4.17 Workaround for multiple classification.** OO languages do not handle this well, so you must use a workaround.

### 4.6.3 Workarounds

Dealing with lack of multiple inheritance is really an implementation issue, but early restructuring of a model is often the easiest way to work around its absence. We list some restructuring techniques below. Two of the approaches make use of ***delegation***, which is an implementation mechanism by which an object forwards an operation to another object for execution. See Chapter 15 for a further discussion of delegation.

- **Delegation using composition of parts.** You can recast a superclass with multiple independent generalizations as a composition in which each constituent part replaces a generalization. This approach is similar to that for multiple classification in the previous section. This approach replaces a single object having a unique ID by a group of related objects that compose an extended object. Inheritance of operations across the composition is not automatic. The composite must catch operations and delegate them to the appropriate part.

  For example, in Figure 4.18 *EmployeeEmployment* becomes a superclass of *FullTimeEmployee* and *PartTimeEmployee*. *EmployeeManagement* becomes a superclass of *Manager* and *IndividualContributor*. Then you can model *Employee* as a composition of *EmployeeEmployment* and *EmployeeManagement*. An operation sent to an *Employee* object would have to be redirected to the *EmployeeEmployment* or *EmployeeManagement* part by the *Employee* class.

  In this approach, you need not create the various combinations (such as *FullTimeIndividualContributor*) as explicit classes. All combinations of subclasses from the different generalizations are possible.

- **Inherit the most important class and delegate the rest.** Figure 4.19 preserves identity and inheritance across the most important generalization. You degrade the remaining generalizations to composition and delegate their operations as in the previous alternative.

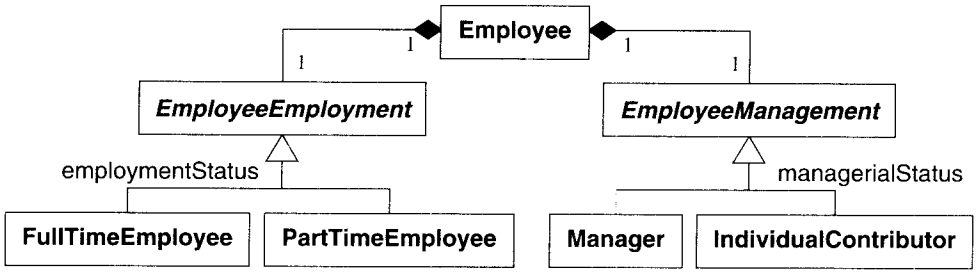


**Figure 4.18 Workaround for multiple inheritance—delegation**

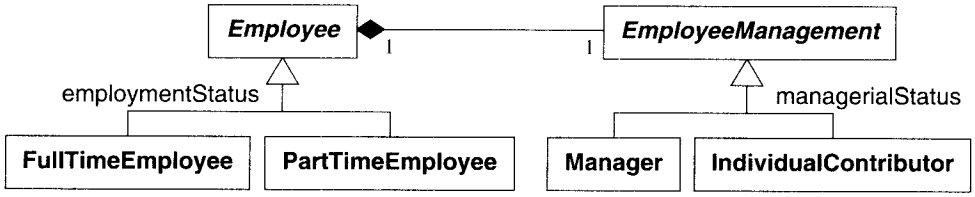


**Figure 4.19 Workaround for multiple inheritance—inheritance and delegation**

- **Nested generalization**. Factor on one generalization first, then the other. This approach multiplies out all possible combinations. For example, in Figure 4.20 under *FullTimeEmployee* and *PartTimeEmployee*, add two subclasses for managers and individual contributors. This preserves inheritance but duplicates declarations and code and violates the spirit of OO programming.

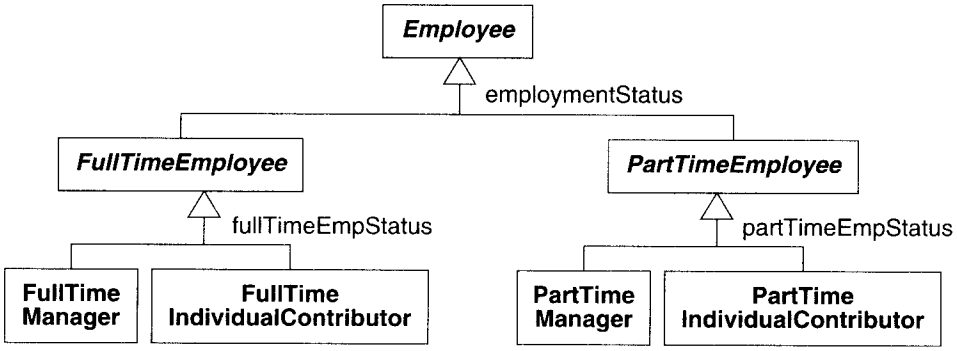


**Figure 4.20 Workaround for multiple inheritance—nested generalization**

Any of these workarounds can be made to work, but they all compromise logical structure and maintainability. There are several issues to consider when selecting the best workaround.

- **Superclasses of equal importance**. If a subclass has several superclasses, all of equal importance, it may be best to use delegation (Figure 4.18) and preserve symmetry in the model.
- **Dominant superclass**. If one superclass clearly dominates and the others are less important, preserve inheritance through this path (Figure 4.19 or Figure 4.20).
- **Few subclasses**. If the number of combinations is small, consider nested generalization (Figure 4.20). If the number of combinations is large, avoid it.
- **Sequencing generalization sets**. If you use nested generalization (Figure 4.20), factor on the most important criterion first, the next most important second, and so forth.
- **Large quantities of code**. Try to avoid nested generalization (Figure 4.20) if you must duplicate large quantities of code.
- **Identity**. Consider the importance of maintaining strict identity. Only nested generalization (Figure 4.20) preserves this.

## 4.7 Metadata

***Metadata*** is data that describes other data. For example, a class definition is metadata. Models are inherently metadata, since they describe the things being modeled (rather than *being* the things). Many real-world applications have metadata, such as parts catalogs, blueprints, and dictionaries. Computer-language implementations also use metadata heavily.

Figure 4.21 shows an example of metadata and data. A car model has a model name, year, base price, and a manufacturer. Some examples of car models are a 1969 Ford Mustang and a 1975 Volkswagen Rabbit. A physical car has a serial number, color, options, and an owner. As an example of physical cars, John Doe may own a blue Ford with serial number *1FABP* and a red Volkswagen with serial number *7E81F*. A car model describes many physical cars and holds common data. A car model is metadata relative to a physical car, which is data.

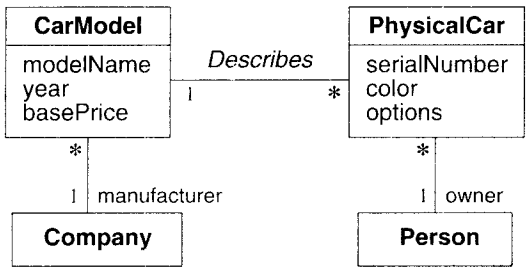


**Figure 4.21 Example of metadata**. Metadata often arises in applications.

You can also consider classes as objects, but classes are meta-objects and not real-world objects. Class descriptor objects have features, and they in turn have their own classes, which

are called ***metaclasses***. Treating everything as an object provides a more uniform implementation and greater functionality for solving complex problems. Languages vary in their accessibility for metadata. Some languages, like Lisp and Smalltalk, let metadata be inspected and altered by programs at run time. In contrast, languages like C++ and Java deal with metadata at compile time but do not make the metadata explicitly available at run time.

## 4.8 Reification

***Reification*** is the promotion of 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.

As an example of reification, consider a database manager. A developer could write code for each application so that it can read and write from files. Instead, for many applications, it is a better idea to reify the notion of data services and use a database manager. A database manager has abstract functionality that provides a general-purpose solution to accessing data reliably and quickly for multiple users.

For another example, consider state-transition diagrams (see the next two chapters). You can use a state-transition diagram to specify control and then implement it by writing the corresponding code. Alternatively, you can prepare a metamodel and store a state-transition model as data. A general-purpose interpreter reads the contents of the metamodel and executes the intent.

Figure 4.22 promotes the *substanceName* attribute to a class to capture the many-to-many relationship between *Substance* and *SubstanceName*. A chemical substance may have multiple aliases. For example, propylene may be referred to as *propylene* and $C_3H_6$. Also, an alias may pertain to multiple chemical substances. Various mixtures of ethylene glycol and automotive additives may have the alias of *antifreeze*.

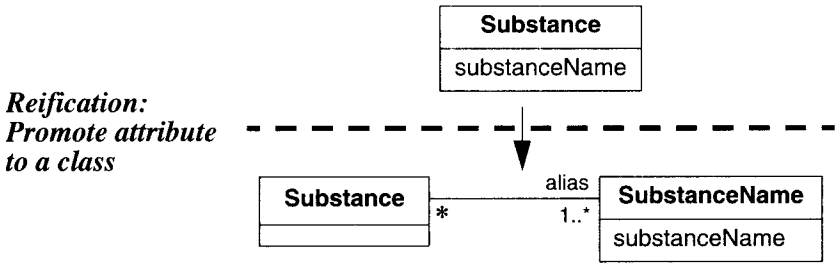


**Figure 4.22 Reification.** Reification is the promotion of something that is not an object into an object and can be helpful for meta applications.

# 4.9 Constraints

A ***constraint*** is a boolean condition involving model elements, such as objects, classes, attributes, links, associations, and generalization sets. A constraint restricts the values that elements can assume. You can express constraints with natural language or a formal language such as the Object Constraint Language (OCL) [Warmer-99].

## 4.9.1 Constraints on Objects

Figure 4.23 shows several examples of constraints. No employee's salary can exceed the salary of the employee's boss (a constraint between two things at the same time). No window can have an aspect ratio (length/width) of less than 0.8 or greater than 1.5 (a constraint between attributes of a single object). The priority of a job may not increase (constraint on the same object over time). You may place simple constraints in class models.

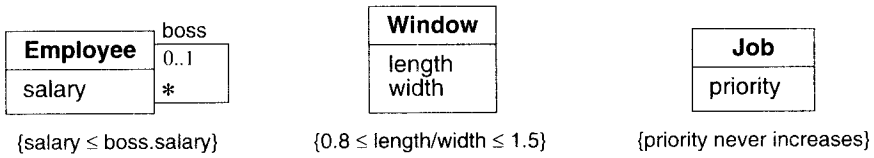


**Figure 4.23 Constraints on objects.** The structure of a model expresses many constraints, but sometimes it is helpful to add explicit constraints.

## 4.9.2 Constraints on Generalization Sets

Class models capture many constraints through their very structure. For example, the semantics of generalization imply certain structural constraints. With single inheritance the subclasses are mutually exclusive. Furthermore, each instance of an abstract superclass corresponds to exactly one subclass instance. Each instance of a concrete superclass corresponds to at most one subclass instance.

Figure 4.16 and Figure 4.17 use a constraint to help express multiple inheritance. The UML defines the following keywords for generalization sets.

- **Disjoint**. The subclasses are mutually exclusive. Each object belongs to exactly one of the subclasses.
- **Overlapping**. The subclasses can share some objects. An object may belong to more than one subclass.
- **Complete**. The generalization lists all the possible subclasses.
- **Incomplete**. The generalization may be missing some subclasses.

### 4.9.3 Constraints on Links

Multiplicity is a constraint on the cardinality of a set. Multiplicity for an association restricts the number of objects related to a given object. Multiplicity for an attribute specifies the number of values that are possible for each instantiation of an attribute.

Qualification also constrains an association. A qualifier attribute does not merely describe the links of an association but is also significant in resolving the "many" objects at an association end.

An association class implies a constraint. An association class is a class in every right; for example, it can have attributes and operations, participate in associations, and participate in generalizations. But an association class has a constraint that an ordinary class does not; it derives identity from instances of the related classes.

An ordinary association presumes no particular order on the objects of a "many" end. The constraint *{ordered}* indicates that the elements of a "many" association end have an explicit order that must be preserved.

Figure 4.24 shows an explicit constraint that is not part of the model's structure. The chair of a committee must be a member of the committee; the *ChairOf* association is a subset of the *MemberOf* association.

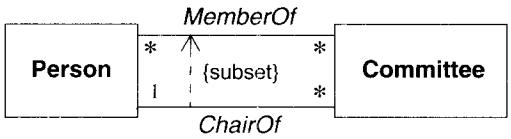


**Figure 4.24 Subset constraint between associations.**

### 4.9.4 Use of Constraints

We favor expressing constraints in a declarative manner. Declaration lets you express a constraint's intent, without supposing an implementation. Typically, you will need to convert constraints to procedural form before you can implement them in a programming language, but this conversion is usually straightforward.

Constraints provide one criterion for measuring the quality of a class model; a "good" class model captures many constraints through its structure. It often requires several iterations to get the structure of a model right from the perspective of constraints. Also, in practice, you cannot enforce every constraint with a model's structure, but you should try to enforce the important ones.

The UML has two alternative notations for constraints. You can either delimit a constraint with braces or place it in a "dog-eared" comment box (Figure 4.26). Either way, you should try to position constraints near the affected elements. You can use dashed lines to connect constrained elements. A dashed arrow can connect a constrained element to the element on which it depends.

## 4.10 Derived Data

A ***derived element*** is a function of one or more elements, which in turn may be derived. A derived element is redundant, because the other elements completely determine it. Ultimately, the derivation tree terminates with base elements. Classes, associations, and attributes may be derived. The notation for a derived element is a slash in front of the element name. You should also show the constraint that determines the derivation.

Figure 4.25 shows a derived attribute. Age can be derived from birthdate and the current date.

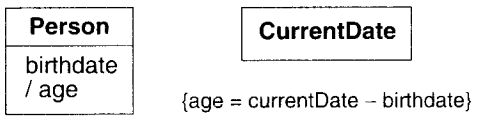


**Figure 4.25 Derived attribute.** A derived attribute is a function of one or more elements.

In Figure 4.26, a machine consists of several assemblies that in turn consist of parts. An assembly has a geometrical offset with respect to machine coordinates; each part has an offset with respect to assembly coordinates. We can define a coordinate system for each part that is derived from machine coordinates, assembly offset, and part offset. This coordinate system can be represented as a derived class called *Offset* related to each part by a derived association called *NetOffset*.

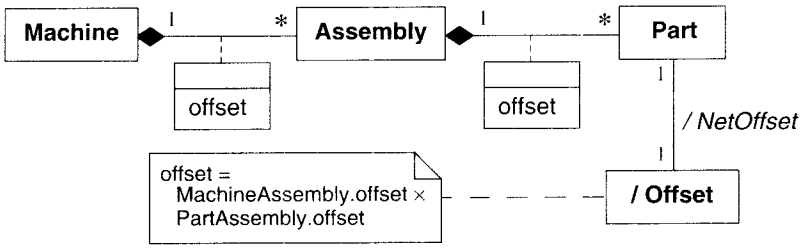


**Figure 4.26 Derived object and association.** Derived data can complicate implementation, so only use derived data where it truly is compelling.

It is useful to distinguish operations with side effects from those that merely compute a functional value without modifying any objects. The latter kind of operation is called a ***query***. You can regard queries with no arguments except the target object as derived attributes. For example, you can compute the width of a box from the positions of its sides. In many cases, an object has a set of attributes with interrelated values, of which only a fixed number of values can be chosen independently. A class model should generally distinguish independent *base attributes* from dependent *derived attributes*. The choice of base attributes is arbitrary but should be made to avoid overspecifying the state of the object.

Some developers tend to include many derived elements. Generally, this is not helpful and clutters a model. You should only include derived elements when they are important application concepts or substantially ease implementation. It can be quite difficult to keep derived elements consistent with the base data, so only use derived elements for implementation where they are clearly compelling.

## 4.11 Packages

You can fit a class model on a single page for many small and medium-sized problems. However, it is often difficult to grasp the entirety of a large model. We recommend that you partition large models so that people can understand them.

A ***package*** is a group of elements (classes, associations, generalizations, and lesser packages) with a common theme. A package partitions a model, making it easier to understand and manage. Large applications may require several tiers of packages. Packages form a tree with increasing abstraction toward the root, which is the application, the top-level package. As Figure 4.27 shows, the notation for a package is a box with a tab. The purpose of the tab is to suggest the enclosed contents, like a tabbed folder.


**Figure 4.27 Notation for a package**. Packages let you organize large models so that persons can more readily understand them.

There are various themes for forming packages: dominant classes, dominant relationships, major aspects of functionality, and symmetry. For example, many business systems have a *Customer* package or a *Part* package; *Customer* and *Part* are dominant classes that are important to the business of a corporation and appear in many applications. In an engineering application we used a dominant relationship, a large generalization for many kinds of equipment, to divide a class model into packages. Equipment was the focus of the model, and the attributes and relationships varied greatly across types of equipment. You could divide the class model of a compiler into packages for lexical analysis, parsing, semantic analysis, code generation, and optimization. Once some packages have been established, symmetry may suggest additional packages.

We can offer the following tips for devising packages.

- **Carefully delineate each package's scope**. The precise boundaries of a package are a matter of judgment. Like other aspects of modeling, defining the scope of a package requires planning and organization. Make sure that class and association names are unique within each package, and use consistent names across packages as much as possible.
- **Define each class in a single package**. The defining package should show the class name, attributes, and operations. Other packages that refer to a class can use a class icon,

a box that contains only the class name. This convention makes it easier to read class models, because a class is prominent in its defining package. Readers are not distracted by definitions that may be inconsistent or misled by forgetting a prior class definition. This convention also makes it easier to develop packages concurrently.

- **Make packages cohesive**. Associations and generalizations should normally appear in a single package, but classes can appear in multiple packages, helping to bind them. Try to limit appearances of classes in multiple packages. Typically no more than 20–30% of classes should appear in multiple packages.

## 4.12 Practical Tips

Here are tips for constructing class models in addition to those from Chapter 3.

- **Enumerations**. When constructing a model, you should declare enumerations and their values, because they often occur and are important to users. Do not create unnecessary generalizations for attributes that are enumerations. Only specialize a class when the subclasses have distinct attributes, operations, or associations. (Section 4.1.1)
- **Class-scoped (static) attributes**. It is acceptable to use an attribute with class scope to hold the extent of a class. Otherwise, you should avoid attributes with class scope because they can lead to an inferior model. You can improve a model by explicitly modeling groups and assigning attributes to them. (Section 4.1.3)
- **N-ary associations**. Try to avoid n-ary associations. Most n-ary associations can be decomposed into binary associations. (Section 4.3)
- **Concrete superclasses**. As a matter of style, it is best to avoid concrete superclasses. Then, abstract and concrete classes are readily apparent at a glance—all superclasses are abstract and all leaf subclasses are concrete. You can always eliminate concrete superclasses by introducing an *Other* subclass. (Section 4.5)
- **Multiple inheritance**. Limit your use of multiple inheritance to that which is essential for a model. (Section 4.6)
- **Constraints**. You may be able to restructure a class model to improve clarity and capture additional constraints. (Section 4.9)
- **Derived elements**. You should always indicate when an element is derived. Use derived elements sparingly. (Section 4.10)
- **Large models**. Use packages to organize large models so that the reader can understand portions of the model at a time, rather than having to deal with the whole model at once. (Section 4.11)
- **Defining classes**. Define each class in a single package and show its features there. Other packages that refer to the class should use a class icon, a box that contains only the class name. This convention makes it easier to read class models and facilitates concurrent development. (Section 4.11)

## 4.13 Chapter Summary

This chapter covers several diverse topics that explain subtleties of class modeling. You will not need these concepts for simple models, but they can be important for complex applications. Remember, application needs should drive the content of any model. Only use the advanced concepts in this chapter if they truly add to your application, either by improving clarity, tightening structural constraints, or permitting expression of a difficult concept.

A data type is a description of values; you must assign every attribute a data type before a model can be implemented. Enumerations are a special data type that constrains the permissible values; enumerated values are often prominent in user interfaces.

Multiplicity is a constraint on the cardinality of a set. It applies to attributes as well as associations. Multiplicity for an association restricts the number of objects related to a given object. Multiplicity for an attribute specifies the number of values that are possible for each attribute instantiation.

You should try to avoid n-ary associations—you can decompose most of them into binary associations. Only use n-ary associations that are atomic and cannot be decomposed. Be aware that most programming languages will force you to promote n-ary associations to classes.

Aggregation is a strong form of association in which an aggregate object is made of constituent parts. Aggregation has the properties of transitivity and antisymmetry that differentiate it from association. Operations on an aggregate often propagate to the constituent parts.

Composition is a form of aggregation with two additional constraints. A constituent part can belong to at most one assembly. Furthermore, once a constituent part has been assigned an assembly, it has a coincident lifetime with the assembly. Composition implies ownership of a part by an assembly.

An abstract class has no direct instances. A concrete class may have direct instances. Abstract classes can define methods in one place for use by several subclasses. You can also use abstract classes to define the signature of an operation, leaving the implementation to each subclass.

Multiple inheritance permits a subclass to inherit features from more than one superclass. Each generalization should discriminate a single aspect You should arrange subclasses into more than one generalization if their superclass specializes on more than one aspect. A subclass may combine classes from different generalizations, or it may combine classes from an overlapping generalization, but it may not combine classes from the same disjoint generalization.

Metadata is data that describes other data. Classes are metadata, since they describe objects. Metadata is a useful concept for two reasons: It occurs in the real world and it is a powerful tool for implementing complex systems. Metadata can be confusing to model, because it blurs the distinction between descriptor and referent. Reification, the promotion of something that is not an object into an object, can be a helpful technique for meta applications.

Explicit constraints on classes, associations, and attributes can increase the precision of a model. Generalization and multiplicity are examples of constraints built into the fabric of class modeling. Derived elements may appear in a model but do not add fundamental information.

A package is a group of classes, associations, generalizations, and lesser packages with a common theme. A package partitions a large model, making it easier to understand and manage.

| | | | |
|---|---|---|---|
| abstract class | concrete class | generalization | package |
| abstract operation | constraint | metadata | propagation |
| aggregation | delegation | multiple inheritance | reification |
| association end | derived element | multiplicity (of an attribute) | scope |
| composition | enumeration | n-ary association | visibility |

**Figure 4.28 Key concepts for Chapter 4**

## Bibliographic Notes

[Rumbaugh-05] explains many subtleties of the UML, some of which we cover in this chapter. [Warmer-99] is the authoritative reference for the Object Constraint Language (OCL).

The previous edition of this book used candidate keys to specify multiplicity for n-ary associations. In this context, a candidate key is a minimal set of association ends that uniquely identifies a link. This edition omits discussion of candidate keys because they are seldom needed for programming. Also, the notion of a *scope* subsumes the terms *class attribute* and *class operation* in the previous edition.

## References

[McAllester-86] David McAllester, Ramin Zabih. Boolean classes. *OOPSLA'87* as *SIGPLAN 22*, 12 (December 1987), 417–424.

[Rumbaugh-88] James E. Rumbaugh. Controlling propagation of operations using attributes on relations. *OOPSLA'88* as *ACM SIGPLAN 23*, 11 (November 1988), 285–296.

[Rumbaugh-05] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual, Second Edition*. Boston: Addison-Wesley, 2005.

[Warmer-99] Jos Warmer and Anneke Kleppe. *The Object Constraint Language*. Boston: Addison-Wesley, 1999.

## Exercises

4.1 (3) The class diagram in Figure E4.1 is a partial representation of the structure of an automobile. Improve it by changing some of the associations to aggregations.

4.2 (4) Figure E4.2 is a partially completed class diagram for an interactive diagram editor. A sheet is a collection of lines and boxes. A line is a series of line segments that connect two boxes. Each line segment is specified by two points. A point may be shared by a vertical and a horizontal line segment in the same line. A selection is a collection of lines and boxes that have been high-

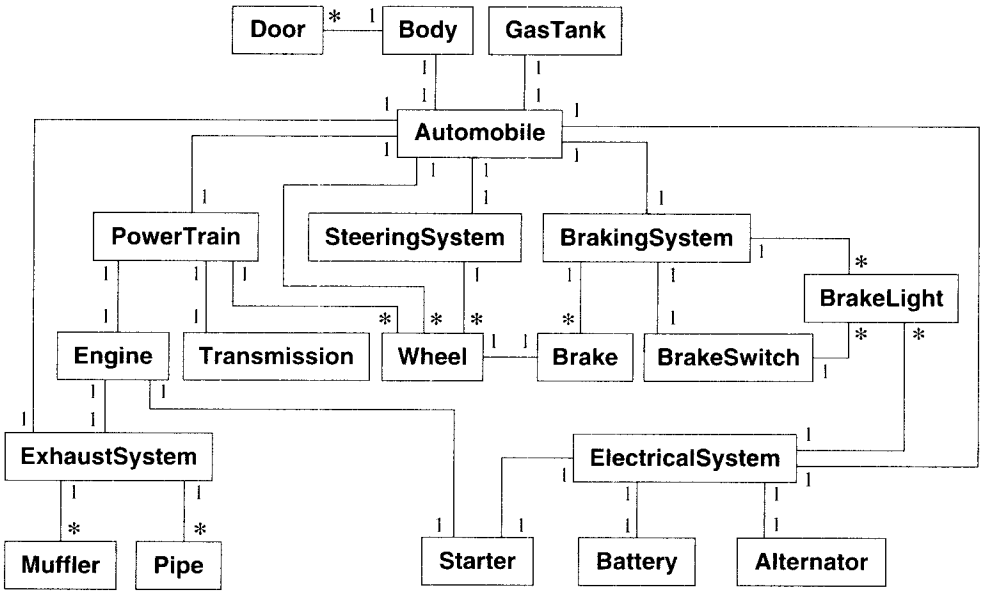


**Figure E4.1 Portion of a class diagram of the assembly hierarchy of an automobile**

lighted in anticipation of an editing operation. A buffer is a collection of lines and boxes that have been cut or copied from the sheet.

As it stands, the diagram does not express the constraint that a line or a box belongs to exactly one buffer or one selection or one sheet. Revise the class diagram and use generalization to express the constraint by creating a superclass for the classes *Buffer*, *Selection*, and *Sheet*. Discuss the merits of the revision.

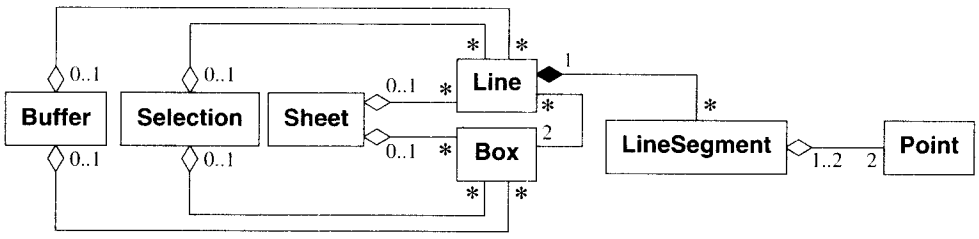


**Figure E4.2 Portion of a class diagram for a simple diagram editor**

4.3 (3) Categorize the following relationships into generalization, aggregation, or association. Beware, there may be n-ary associations in the list, so do not assume every relationship involving three or more classes is a generalization. Explain your answers.

a. A country has a capital city.

b. A dining philosopher uses a fork.

c. A file is an ordinary file or a directory file.

d. Files contain records.
e. A polygon is composed of an ordered set of points.
f. A drawing object is text, a geometrical object, or a group.
g. A person uses a computer language on a project.
h. Modems and keyboards are input/output devices.
i. Classes may have several attributes.
j. A person plays for a team in a certain year.
k. A route connects two cities.
l. A student takes a course from a professor.

4.4 (7) Prepare a class diagram for a graphical document editor that supports grouping. Assume that a document consists of several sheets. Each sheet contains drawing objects, including text, geometrical objects, and groups. A group is simply a set of drawing objects, possibly including other groups. A group must contain at least two drawing objects. A drawing object can be a direct member of at most one group. Geometrical objects include circles, ellipses, rectangles, lines, and squares.

4.5 (7) The following is a partial taxonomy of rotating electrical machines. Electrical machines may be categorized for analysis purposes into alternating current (ac) or direct current (dc). Some machines run on ac, some on dc, and some will run on either. A few examples of electrical machines include large synchronous motors, small induction motors, universal motors, and permanent magnet motors. Most motors found in the home are usually induction or universal.

An ac machine may be synchronous or induction. Universal motors are typically used where high speed is needed, such as in blenders or vacuum cleaners. They will run on either ac or dc. Permanent-magnet motors are frequently used in toys and will work only on dc.

Prepare a class diagram showing how the categories and the machines just described relate to one another. Use multiple inheritance where it is appropriate to do so.

4.6 (7) Revise the class diagram that you prepared for the previous exercise to eliminate use of multiple inheritance.

4.7 (8) Prepare a metamodel that supports only the following UML concepts: class, attribute, association, association end, multiplicity, class name, and attribute name. Use only these constructs to build your metamodel.

4.8 (8) Prepare an object diagram of the metamodel you prepared in the previous diagram. Treat the metamodel as a class diagram that can be represented by instances of the classes of the metamodel.

4.9 (5) Use generalization to revise your answer from Exercise 4.7 so that an attribute belongs to either a class or an association, but not both at the same time.

4.10 (7) Figure E4.3 is a portion of a metamodel that describes generalization. A generalization is associated with several generalization roles, which are the roles that classes play in generalizations. Role type is either subclass or superclass. Does this model support multiple inheritance? Explain your answer.

4.11 (8) Describe how to find which class is the superclass of a generalization using the metamodel in Figure E4.3. Revise the metamodel to simplify the query. Describe how to determine the superclass of a generalization using your revised metamodel. Make sure that your revised metamodel supports multiple inheritance. Write OCL queries for Figure E4.3 and your model to find a superclass, given a generalization.

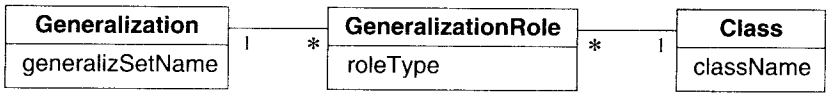


**Figure E4.3 Metamodel of generalization**

4.12 (7) How well does the metamodel in Figure E4.3 enforce the constraint that every generalization has exactly one superclass? Revise it to improve the enforcement of the constraint.

4.13 (7) Figure E4.3 is a metamodel that describes class models such as in Figure E4.4. Prepare an object diagram using the classes from the metamodel to describe the model in Figure E4.4.

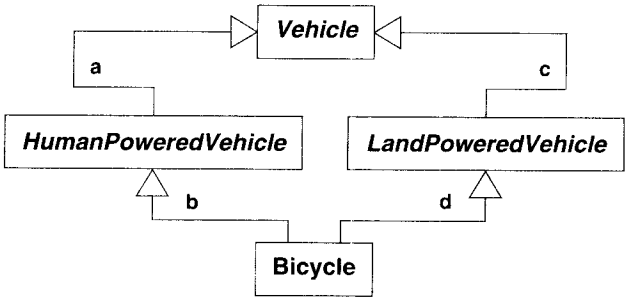


**Figure E4.4 Class diagram with multiple inheritance**

4.14 (6) Prepare a portion of a class diagram for a library book checkout system that shows the late charges for an overdue book as a derived attribute.

4.15 (10) Prepare a metamodel of Backus-Naur (BNF) representations of computer languages. A compiler-compiler (such as the UNIX program YACC) could use the model. The compiler-compiler accepts these representations in graphical form as input and produces a compiler for the represented language.

Figure E4.5 shows an example of a Backus-Naur form that the compiler-compiler will accept. Rectangles denote nonterminals, and circles or rectangles with rounded corners denote terminals. Single characters are in circles, and sequences of several characters are in rounded rectangles. Arrows indicate the direction of flow through the diagram. Where several directed paths diverge, it is permissible to take any one of them. The name of the nonterminal being described appears at the beginning of its representation.

4.16 (7) Prepare a simple class model, sufficient for representing recipes. Use the recipe in Figure E4.6 as a basis. This exercise is an example of reification. In one sense the tasks of a recipe could be operations; in another sense they could be data in a class model.

4.17 (9) Extend your class model of recipes to handle alternate ingredients. For example, some lasagna recipes allow cottage cheese to be substituted for ricotta cheese.

4.18 (8) The North American Securities Administrators Association (NASAA, www.nasaa.org) seeks to protect investors and educate them about trading in securities. NASAA recommends

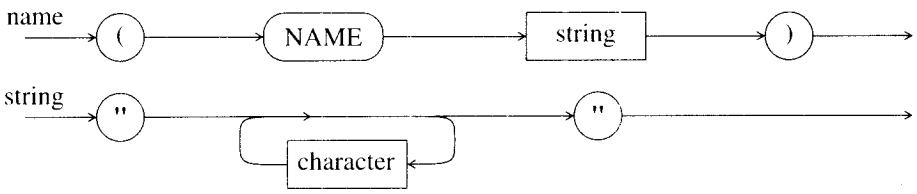


**Figure E4.5 Portion of a BNF diagram**

**Lasagna**

| | |
|---|---|
| 2.5 tbsp. salad oil for browning | 1 tsp. oregano |
| 1 cup minced onion | .5 box lasagna noodles |
| 1 clove garlic | 1 lb. ricotta cheese |
| 1 lb. ground beef | 1 cup grated mozzarella cheese |
| 2 tsp. salt | .5 cup parmesan cheese |
| 3.5 cups whole tomatoes (large can) | 2 cans tomato paste |

Cook onion, clove garlic, ground beef, 1 tsp. salt in salad oil until meat is browned. Add tomatoes, tomato paste, 1 tsp salt, oregano, and simmer, covered, 1 hour until thick. Cook noodles 15 minutes in water until tender. Drain and blanch. Butter 12×8 inch pan and place in layers of noodles, sauce, mozzarella, ricotta cheese, and parmesan. Bake at 350 degrees for 45 to 60 minutes.

**Figure E4.6 A simple recipe**

that investors take notes when talking to a broker using the form in Figure E4.7. (Use multiple forms when the broker makes multiple recommendations in a call.) Suppose that it is desirable to automate this form with software. Prepare a class model for the form.

4.19 (9) Prepare a model for words in a dictionary. Include the following: alternative spellings, antonyms, dictionary, grammar type (noun, verb, adjective, adverb), historical derivation, hyphenation, meanings, miscellaneous comments, prioritization by frequency of use, pronunciation, and synonyms.

Some sample definitions are as follows (from *Webster's New World Dictionary*):

- **been** (bin; *also, chiefly Brit.*, bēn &, *esp. if unstressed*, ben), pp. of **be**.
- **kum·quat** (kum′kwot), *n.* [< Chin. *chin-chü*, golden orange], 1. a small, orange-colored, oval fruit, with a sour pulp and a sweet rind, used in preserves, 2. the tree that it grows on. Also sp. **cumquat**.
- **lac·y** (lās′i), *adj.* [-IER, -IEST], 1. of lace. 2. like lace; having a delicate open pattern. — **lac′i·ly**, *adv.* —**lac′i·ness**, *n.*
- **Span·ish** (span′ish), *adj.* of Spain, its people, their language, etc. *n.* 1. the Romance language of Spain and Spanish America. 2. the Spanish people.

Figure E4.8 shows a partial answer to the problem showing classes and relationships. Add attributes and ordering to the associations where appropriate.

The *RelatedWord*, *Synonym*, and *Antonym* associations are not quite right and have a problem. Comment on them.

Date _____ Time _____

☐ Call made ☐ Call received ☐ Meeting Location _____

Name of Broker _____ Phone _____

Broker's Firm _____ Phone _____

Broker's CRD No. _____ ☐ Obtained CRD Report

**Investment Recommendation**

☐ Buy ☐ Sell

Name of Security _____

Reasons for recommendation _____

How does this meet my investment objectives? _____

What are the risks? _____

I asked to receive written information about the investment before making a decision.

☐ Yes ☐ No

I will get:

☐ a prospectus
☐ an offering memorandum
☐ most recent Annual Report
☐ most recent quarterly or interim reports
☐ research reports
☐ other information

**Proposed Trade**

Number of shares/units _____

Price per share $ _____

Total cost $ _____

commission _____

**My instructions**

☐ Do nothing ☐ Buy ☐ Sell

Number Price

$ _____

Notes _____

Notes made by: _____

**Figure E4.7 NASAA form for broker calls**

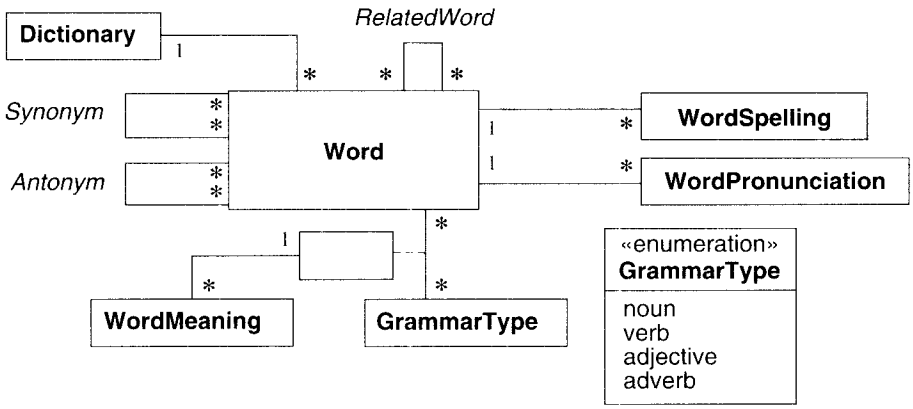


**Figure E4.8 Partial model for words in a dictionary**

# 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