2

Object-Oriented Modeling and Design for Database Applications

MICHAEL BLAHA
OMT Associates Inc.
Chesterfield, Missouri

WILLIAM PREMERLANI
General Electric Research and Development Center
Schenectady, New York

CI-5311
OMT estático
y algo de calidad del
esquema.

Prentice Hall
Upper Saddle River, New Jersey 07458

1

Introduction

First we discuss the purpose of object-oriented software engineering and the benefits of modeling. Then we explain object orientation's contribution to database applications and introduce the Object Modeling Technique (OMT) methodology. We explain the relationship of this book to our prior book [Rumbaugh-91] in the Bibliographic Notes at the end of the chapter.

1.1 WHY OBJECT ORIENTATION?

Object orientation is a strategy for organizing systems as collections of interacting objects that combine data and behavior. It applies to many technology areas, including hardware, programming languages, databases, user interfaces, and software engineering. An object-oriented approach to software development is superior to a procedural approach. For example, an object-oriented approach lets you organize database structures and program functionality around the same paradigm. Databases and programs can be developed together for ease of conceptualization, implementation, maintenance, and potential reuse.

The object-oriented philosophy creates a powerful synergy throughout the development life cycle by combining abstraction, encapsulation, and modularity. *Abstraction* lets you focus on essential aspects of an application while ignoring details. It also lets you preserve design freedom until later stages of development, when you can make more informed decisions. The ability to abstract is probably the most important skill required for object-oriented development, a skill that you can learn by studying this book and working the exercises. *Encapsulation* separates external specification from internal implementation. External specification must be addressed early in development; internal implementation can largely be deferred. *Modularity* promotes coherence, understandability, and symmetry by organizing a system into groups of closely related objects.

The ability to promote abstraction makes an object-oriented approach particularly appealing for software engineering. ***Software engineering*** is a systematic approach to software

development that emphasizes thorough conceptual understanding prior to design and coding. Software is most malleable during the early stages of development; errors and oversights are easier to resolve during early stages of software development than once an application is cast in programming and database code. Thorough understanding of a problem can improve software reliability, increase flexibility, reduce development time, and reduce cost.

1.2 OBJECT-ORIENTED SOFTWARE ENGINEERING

You can perform object-oriented software engineering by constructing models. A ***model*** is an abstraction of some aspect of a problem; we express models with various kinds of diagrams. With an object-oriented approach you begin by describing relevant aspects of the real world and gradually elaborate and refine the description until you realize a working system. Thus models not only provide a basis for analyzing requirements but provide the genesis for design and implementation. There are additional advantages of models:

- **Reduced life cycle cost**. The clear documentation helps maintenance and allows more software reuse to occur—from code libraries, from related projects, and within a project. Code becomes smaller in size, more focused, more consistent, and easier to manipulate. You can use tools to generate code.
- **Better quality**. Models provide a language that enables deep thinking about a design. Developers can concentrate on the important issues for a system and defer attention to syntactic details. When a design is formally expressed and subjected to scrutiny by others, the resulting software is more likely to be robust and correct. A modeling language allows a more precise expression of ideas than does a natural language.
- **Faster time to market**. You can organize large projects into work units that can be assigned to different development teams. The clearly defined boundaries between portions of a system let new work occur with minimal impact on previous or parallel work. Software reuse also helps to shorten delivery time.
- **Communication**. Models bring important names and application concepts to the fore so they can be defined and understood by all parties. Models promote communication between developers and customers by separating deep conceptual issues from distracting implementation details.
- **Extensibility**. Software organized about an object-oriented theme parallels the real world and is flexible with respect to changes in requirements; to a large extent the software may be extended for new requirements without disrupting solutions to existing requirements. In contrast software that is decomposed into arbitrary functions (the procedural approach) is brittle and often difficult to evolve.
- **Traceability**. Software development is a seamless process. With object-oriented development the models derived from analysis of customer requirements are carried forward and permeate subsequent development steps. With tool support you can trace the impact of the requirements on the actual delivered system.

- **Better selection**. Models are useful even when software is purchased from a vendor rather than developed. Models can aid understanding of vendor software and enrich the basis for selecting the best product. You can choose the best product not only on the basis of functionality, user interface, and cost but also on the soundness of its supporting infrastructure.

Philosophically, we cannot think of any disadvantage of modeling. Modeling only consumes a modest amount of time; this is easily outweighed by the aforementioned advantages.

We are aware of one practical impediment to modeling—modeling can be difficult to learn. Many persons have difficulty with abstraction and the indirection of realizing an application through models. Our best advice is that developers should pore over this book and work the exercises to become proficient at modeling. You have to practice modeling to learn it. Developers can also benefit from university courses, commercial training, and expert consulting help as they experience modeling for their first few projects.

Do not confuse modeling with programming; object-oriented modeling is not the same as object-oriented programming. Object-oriented programming only concerns latter stages of the software life cycle. Object-oriented software engineering permeates the life cycle and even applies with implementation in a non-object-oriented language. The economic impact of object-oriented software engineering is much broader than that of object-oriented programming.

1.2.1 Benefits of OO Software Development to Database Applications

Object-oriented (OO) modeling is especially helpful for database applications. We give some specific advantages in this section. We assume that most readers know basic database terms, but for those who don't, we first define a few of the more important terms.

A ***database*** is a permanent, self-descriptive repository of data that is stored in one or more files. Self-description is what sets apart a database from ordinary files. A database contains the data structure or ***schema***—description of data—as well as the data. The premise of databases is that the data structure is expected to be relatively static while the actual data may rapidly evolve. Most database systems operate as a closed world; if a fact is not found in the database, the fact is assumed to be false.

A ***database management system*** (***DBMS***) is the software for managing access to a database. DBMSs are intended to provide generic functionality for a wide variety of applications. One of the broad objectives of object-oriented technology is to promote software reuse; for data-intensive applications DBMSs can substitute for much application code. You are achieving reuse when you can use generic DBMS code, rather than custom-written application code. A DBMS has one or more ***query languages***; a query language provides commands for reading and writing data interactively or via a program.

A ***relational database*** is a database in which the data is logically perceived as tables. A ***relational DBMS*** manages tables of data and associated structures that increase the functionality and performance of tables. ***Normal forms*** are guidelines for relational database design that increase the consistency of data. As tables satisfy higher normal forms, they are more likely to store correct data.

An ***object-oriented database*** can be regarded as a persistent store of objects created by an object-oriented programming language. With an ordinary programming language, objects cease to exist at program termination; with an object-oriented database, objects persist beyond the confines of program execution. An ***object-oriented DBMS*** manages the data, programming code, and associated structures that constitute an object-oriented database. In contrast to relational DBMSs, object-oriented DBMSs vary widely in their syntax and capabilities. The Object Data Management Group (ODMG) standards effort [Cattell-96] is gradually resolving this excessive variation and causing a convergence in OO-DBMS syntax and features.

Object-oriented modeling provides benefits for database applications in addition to the general benefits mentioned earlier.

- **One development paradigm**. Object-oriented models provide a uniform abstraction for the design of both programming code and database code. Object-oriented models map naturally to all major languages and the standard types of DBMSs. Unlike a procedural approach, you can apply the same paradigm throughout analysis, design, and implementation. You need not convert from one paradigm to another as software development proceeds.
- **Improved quality of data**. Rigorous modeling improves the quality of the data in a database. You can weave many constraints into the database structure. Modeling lets you clearly see nonstructural constraints, so you can enforce them with the programming code that accompanies the database. An object-oriented model naturally leads to a normalized relational database schema. (Chapter 13 presents mapping rules that you can apply to an object-oriented model to yield a normalized relational database schema.)
- **Better performance**. A well-conceived model simplifies database tuning. A database that crisply corresponds to the real world will simplify expression of queries and facilitate a rapid DBMS response.
- **Faster development**. Careful modeling is required to exploit DBMS capabilities fully. A clear and consistent database structure can simplify programming code. Furthermore, skillful developers can often substitute powerful DBMS commands for tedious, custom-written code.
- **Less debugging**. Flawed models cause errors in the corresponding applications. A poorly conceived database structure impairs the ability of the database to detect application errors. Database constraints can assure quality by providing a means to enforce integrity apart from application code.
- **Easier migration**. Software comes and goes as new systems are deployed and old systems become obsolete. However, data must carry forward and be preserved across systems. Sound models can survive system obsolescence and facilitate migration of legacy data.
- **Easier integration**. Integration of stand-alone systems can reduce human transcription of data and broaden the queries that can be answered. However, for integration to succeed, system architects must thoroughly understand the applications—purchased software, legacy systems, and newly developed software. Object-oriented models provide a uniform representation that facilitates understanding and integration.

1.3 THE OMT METHODOLOGY

There are many software development approaches that utilize object-oriented models, but not all of them are suitable for database applications. Some approaches lack sufficient abstraction and are tied too tightly to implementation details. Others are too programming oriented and fail to emphasize data structures and constraints adequately, which are particularly important to database applications.

Our approach, OMT, stresses the importance of models and uses models to achieve abstraction. We crisply differentiate the analysis stage, which focuses on the real world, from the design stage, which addresses the particular details of the computer resources. As such, the OMT methodology can apply to various implementation targets, including files, relational databases, and object-oriented databases. The OMT models are built around descriptions of data structure, behavior, and constraints. The OMT methodology addresses both transaction-processing systems (high volume, simple processing) and decision-support systems (a rich variety of unpredictable queries). The Bibliographic Notes at the end of this chapter comment on the relationship of this book to our prior book [Rumbaugh-91].

The OMT approach builds on past work. From the object-oriented programming community we have incorporated the notion of objects as coherent entities with identity, state, and behavior. Objects may be organized by their similarities and differences enabling the use of inheritance and polymorphism. From information modeling we have adopted the notion of entities that are connected by relationships. Information models tend to be *declarative* (stated as structural properties and constraints) rather than *imperative* (stated as a series of programming statements). We have also included ideas from software engineering, such as the notion of a formal process for software development with emphasis placed on early conceptual stages rather than later implementation details. Misunderstandings and flaws that are detected early are much easier and less costly to correct.

The OMT methodology emphasizes declarative specification of information. Declaration lets you cleanly capture requirements; imperative specification runs the risk of prematurely descending into design. Declarative specifications tend to be more concise and allow machine-based optimization. Many DBMSs provide direct implementation support for declarative specification.

1.3.1 OMT Development Process

A bonafide methodology consists of both process and concepts. A clear process must guide the practitioner through system development. The process is supported by underlying concepts and a notation for expressing the concepts. The OMT development process comprises the following sequence of steps. (The process preview in Chapter 6 expands the description of these steps.) Much iteration occurs within the steps and some iteration occurs across the steps.

- **Conceptualization**. Software development begins with business analysts or users conceiving an application and formulating tentative requirements. Conceptualization is often stimulated by business process reengineering (BPR)—critical examination of busi-

ness processes and their impact on profitability. BPR identifies opportunities for new systems, specifies their purpose, and determines their scope. BPR is outside the scope of this book.

- **Analysis**. The requirements formed during conceptualization are scrutinized and rigorously restated by constructing real-world models. The goal of analysis is to specify *what* needs to be done without constraining *how* it is done.
- **System design**. The development team devises a high-level strategy—the system architecture—for solving the application problem. Policies must also be established that will serve as a default for the subsequent, more detailed portions of design.
- **Detailed design**. Real-world analysis models are augmented and transformed into a form amenable to computer implementation. We perform this shift in the models without descending into the particular details of the target database or programming language. We have renamed this step from *object design* in [Rumbaugh-91].
- **Implementation**. The design is translated into the actual programming language and database code. This step is straightforward, because much work has already been performed during analysis and design.
- **Maintenance**. The development documentation and the traceability from the models through the code facilitate subsequent maintenance. Treatment of maintenance, testing, and project management is outside the scope of this book.

The OMT methodology supports multiple development styles. You can use OMT as a waterfall approach performing the phases of analysis, design, and implementation in strict sequence for the entire system. More often, we adopt an iterative development strategy. First we develop the nucleus of a system—analyzing, designing, implementing, and delivering working code. Then we grow the scope of the system, adding properties and behavior to existing objects, as well as adding new kinds of objects.

Conceptualization and analysis are largely independent of the target platform. You must first understand an application before you can proceed, regardless of whether the application is being implemented with a programming language, database, or some other means. In contrast, design, implementation, and maintenance are heavily influenced by the target platform—in the case of this book, a database.

1.3.2 Entity-Based versus Attribute-Based Development

As we have said, the OMT approach focuses on objects that are coherent entities. This is in contrast to the attribute-based approach traditionally associated with relational database applications. We believe the distinction is purely philosophical, however, and not at all intrinsic to relational databases themselves.

In an entity-based approach, originated by [Chen-76], you note entities from the real world, describe them, and observe relationships among them. The OMT methodology adopts an entity-based approach. In contrast, with an attribute-based approach you list attributes that are meaningful to an application and organize them into groups.

In practice, entity-based development is clearly superior to attribute-based development. First, most applications have an order of magnitude fewer entities than attributes, so an entity-based approach is more tractable. Second, attribute-based development requires that you check normal forms—to ensure that attributes are combined in a reasonable manner so update anomalies are unlikely to occur. These checks can be tedious and confusing. With an entity-based approach attributes are introduced as part of describing real-world objects. To the extent that you directly describe objects and do not intermix different things, the anomalies addressed by normal forms do not arise. This real-world basis also makes an entity-based model more extensible, understandable, and tunable for fast performance. Finally, an entity-based approach is more likely to be compatible with the development approach for programming code and the overall system.

1.3.3 The Three OMT Models

The creator of a modeling language must confront the trade-off of expressiveness and understandability. A simple language is coherent but may fail to capture fully the nuances of a problem. A complex language permits more precise specification but may inhibit broad understanding. We have attempted to strike a balance between modeling power and simplicity with the OMT notation. As an example of the richness of the OMT notation, you can specify much programming behavior with simple navigation of an object model. Yet the OMT notation is understandable, as we have learned through experience with users on various projects.

As Figure 1.1 shows, the OMT methodology uses three complementary models to express aspects of a problem and its solution. Each model presents a perspective of the system, which by itself is incomplete. The models combine to describe a system fully. Each model contributes important information to the development process, but the models do not have equal importance for an individual application. Database applications, for example, tend to rely heavily on the object model.

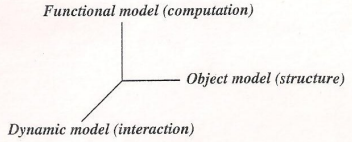


Figure 1.1 The three OMT models

We briefly describe the models here in preparation for more elaborate discussion in subsequent chapters.

- **Object model**. The object model characterizes the static structure of things (objects). It looks at structure in terms of groups of analogous objects (classes), their similarities and

differences (generalization), and their important relationships with one another (associations). The object model defines the context for software development—the universe of discourse. An object model is expressed with one or more class diagrams. The object model is important for database applications because it concisely describes data structure and captures structural constraints.

- **Dynamic model**. The dynamic model describes temporal interactions between objects —various stimuli that occur and the response of objects to the stimuli. The objects in a class pass through states with similar responses to events. State diagrams express an aspect of behavior that is shared by the objects in a class. There is one state diagram for each class with significant dynamic behavior; the collection of interacting state diagrams constitutes the dynamic model.

  The typical database application is dominated by storage and retrieval, and the dynamic model is seldom important. (In contrast, the dynamic model is quite significant for many other kinds of applications, such as user interfaces and control applications.) Given that the focus of this book is on data management applications, we will omit treatment of the dynamic model. For a discussion of the dynamic model, see [Rumbaugh-91]. The full OMT methodology still includes the dynamic model, but we do not discuss the dynamic model in this book.

- **Functional model**. The functional model defines the computations that objects perform—how output values are computed from input values. The functional model specifies operations that arise within the object and dynamic models. We recommend several notations for expressing the functional model, though we emphasize pseudocode enhanced with the object navigation notation. (This is a departure from [Rumbaugh-91], which emphasizes data flow diagrams.) We believe it is beneficial to have more than one notation and to be able to mix them. The primary importance of the functional model for database applications is in showing navigation of the object model.

1.4 ORGANIZATION OF THIS BOOK

This chapter introduces basic concepts for object orientation, software engineering, and databases. The remainder of the book consists of four parts: concepts and notation, analysis and design, implementation, and large systems issues. The appendixes provide a glossary of terminology and summarize the functional modeling notation.

Part 1 (Chapters 2 through 5) describes OMT concepts and notation. We explain the object model in great detail because database applications tend to have rich data structure. We also cover the functional model, but omit the dynamic model. Our experience has been that data management applications seldom involve a significant dynamic model.

Part 2 (Chapters 6 through 11) presents the OMT analysis and design process. We introduce the financial portfolio manager case study and thread it throughout Parts 2 and 3.

Part 3 (Chapters 12 through 17) shows how to implement an OMT design for a variety of platforms: files, relational databases, and object-oriented databases (ObjectStore). We emphasize databases in this book, but also cover files because files are a reasonable option for

managing data for many applications. We fully implement the portfolio manager case study for the MS-Access relational database. You can find the detailed code at our Web site (www.omtassociates.com).

Part 4 (Chapters 18 through 20) concludes the book with treatment of several large systems issues: distributed databases, integration of systems of applications, and reverse engineering of legacy databases.

This book is self-contained. However, the organization of the book deliberately parallels that of our past book [Rumbaugh-91], so you can see more clearly how database considerations affect the OMT development process.

All major chapters of this book contain exercises with indicated difficulty levels. The exercises bring out subtleties you may miss in the text. If you want to learn this material well, you should work the exercises.

1.5 FONT CONVENTIONS

We use several font conventions throughout this book. We begin most bullet items with a phrase in bold font to denote the theme. We italicize words that are modeling elements or for emphasis. We use bold, italic font to call attention to defined terms that are also listed in Appendix A.

1.6 CHAPTER SUMMARY

Object orientation is a strategy for organizing systems, especially hardware and software systems, as collections of interacting objects that combine data and behavior. Software engineering is a systematic approach to software development that emphasizes thorough conceptual understanding prior to design and coding. This book concerns object-oriented software engineering, the intersection of object orientation and software engineering. You can perform object-oriented software engineering by constructing models that abstract various aspects of a problem. You describe relevant aspects of the real world and gradually elaborate and refine the description until you realize a working system.

The OMT methodology is our approach for object-oriented software engineering. Any bonafide methodology consists of both process and concepts. A clear process must guide the practitioner through system development. The process is supported by underlying concepts and a notation for expressing the concepts. The OMT development process comprises the major steps of analysis, design, and implementation. During analysis the developer scrutinizes and rigorously restates requirements by constructing real-world models; the goal of analysis is to specify what needs to be done without constraining how it is done. During design the developer must shift focus from the real world toward the underlying computer resources. Implementation consists of relatively straightforward translation of the design into programming and database code.

The OMT methodology uses three complementary models that support the process. The object model characterizes the static structure of a system in terms of groups of analogous

objects (classes), their similarities and differences (generalization), and their important relationships with one another (associations). The dynamic model describes temporal interactions between objects—various stimuli that occur and the response of objects to the stimuli. The functional model defines the computations that objects perform—how output values are computed from input values. Database applications seldom involve a significant dynamic model, so we omit coverage of the dynamic model from this book.

Figure 1.2 lists key concepts that we have discussed in this chapter.

abstraction	implementation
analysis	object model
conceptualization	object orientation
database	OMT methodology
database management system (DBMS)	schema
detailed design	software engineering
dynamic model	system design
functional model	

Figure 1.2 Key concepts for Chapter 1

BIBLIOGRAPHIC NOTES

The OMT object model is essentially an extended Entity-Relationship approach. [Chen-76] is the seminal paper on the Entity-Relationship approach to modeling. The OMT dynamic model is taken from [Harel-87] with some minor adjustments.

The general OMT methodology—including notation, concepts, and an analysis process that spans programming and database applications—is presented in an earlier book [Rumbaugh-91]. However, the treatment of databases, especially the design and implementation process, in [Rumbaugh-91] is incomplete. [Rumbaugh-96] extends the OMT methodology, mostly for programming applications.

This book is a sequel that supersedes [Rumbaugh-91] for data management applications. We have introduced additional advanced object modeling concepts and an entire chapter on metamodeling. We have also reworked the functional model. Nevertheless, our primary contribution lies in the areas of design, implementation, and large system issues. We have completely rewritten the design and implementation portions to address properly applications that are data intensive. Database applications require custom treatment because they are much different from programming applications. Database applications tend to have a much larger and more complex data structure and less rich behavior compared to programming applications. The exercises in this book complement those in [Rumbaugh-91].

Booch, Rumbaugh, and Jacobson have developed a new notation that unifies the Booch, OMT, and Objectory notations [UML-98]. They have made several changes in their notation that are pertinent to databases, and we have adopted them. Our emphasis in this book is on fleshing out the OMT process for data management applications, not on creating new notation. The UML authors are addressing programming applications; we are addressing database applications. The union of their efforts and our efforts constitutes the next generation of the OMT methodology.

In general, there are two major camps of object-oriented approaches to software development. The information-centric approaches start by building an enhanced Entity-Relationship model and then add behavior. OMT, Shlaer-Mellor [Shlaer-88], and Martin-Odell [Martin-92] are information-centric methodologies. In contrast, the behavior-centric approaches start by describing the behavior of a system and then develop an information model. Object-Oriented Software Engineering (OOSE) [Jacobson-92] and Requirements Driven Design (RDD) [Wirfs-Brock-90] illustrate the behavior-centric approach. Information-centric approaches tend to be declarative (organized about data structure), while behavior-centric approaches tend to be imperative (programming oriented).

We emphasize that software development is a seamless process; OMT models are gradually elaborated and optimized as the focus shifts from analysis to design to implementation. Some methodologists, such as [Shlaer-88], adopt an alternate viewpoint—shifting notations when proceeding from analysis to design. Our experience has been that a shift in notation is unnecessary, causes confusion, and can lose information.

REFERENCES

[Cattell-96] R Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. San Mateo, California: Morgan Kaufmann, 1996.

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

[Harel-87] David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming 8* (1987), 231–274.

[Jacobson-92] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Reading, Massachusetts: Addison-Wesley, 1992.

[Martin-92] James Martin and James Odell. *Object-Oriented Analysis and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1992.

[Rumbaugh-91] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[Rumbaugh-96] James Rumbaugh. *OMT Insights*. New York: SIGS Books, 1996.

[Shlaer-88] Sally Shlaer and Stephen J. Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, New Jersey: Yourdon Press, 1988.

[UML-98] The following books are planned for the Unified Modeling Language:
Grady Booch, James Rumbaugh, and Ivar Jacobson. *UML User's Guide*. Reading, Massachusetts: Addison-Wesley.
James Rumbaugh, Ivar Jacobson, and Grady Booch. *UML Reference Manual*. Reading, Massachusetts: Addison-Wesley.
Ivar Jacobson, Grady Booch, and James Rumbaugh. *UML Process Book*. Reading, Massachusetts: Addison-Wesley.

[Wirfs-Brock-90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

PART 1: MODELING CONCEPTS

2

Basic Object Modeling

Part 1 presents the object and functional models. We define notation, explain the ideas embodied by the notation, and present many examples. We have made several changes in the original OMT notation to make it consistent with the work of Booch, Rumbaugh, and Jacobson [UML-98]. We summarize these changes at the end of each chapter.

The OMT notation spans the development lifecycle; the models that are developed during analysis carry forward to design and implementation. As software development proceeds, the developer elaborates and optimizes the OMT models. The emphasis in Part 1 is on analysis. Database applications require some additional notation for design and implementation; we present this additional notation in Parts 2 and 3.

We explore the object model in greater detail than the functional model and omit the dynamic model because database applications tend to be dominated by the object model. The dynamic model is seldom important for such applications, although it is integral to the OMT methodology for other purposes, such as user interfaces and control applications.

The object model describes a system's data structure, its objects and their relationships. Object models are built from classes that relate to each other through associations and generalizations. The object model defines the context for software development and provides the lattice to which behavior is attached. Chapter 2 describes object modeling concepts that frequently occur and that are sufficient for modeling many applications. Chapter 3 addresses additional aspects of object modeling that are required for advanced applications. Chapter 4 is very advanced and discusses object metamodeling.

The OMT approach places greater emphasis on data structure and constraints than do most other object-oriented methodologies. With the object model, there are often alternative ways to represent an application. (See Exercise 2.13.) It is important to exploit the richness of OMT modeling and capture data structure precisely. A thorough object model enforces many constraints and makes it easier to navigate among objects. The richness of the object model obviates much of the algorithmic code required by other methodologies.

2.1 OBJECT AND CLASS CONCEPTS

2.1.1 Objects

An ***object*** is a concept, abstraction, or thing that has meaning for an application. Objects often appear as proper nouns or specific references in problem descriptions or in 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.

Each object exists and can be identified. For example, even though two persons may have the same name, they are separate. ***Identity*** is "that property of an object which distinguishes each object from all others" [Khoshafian-86]. During analysis, you can take for granted that objects have identity. During design, you must choose an approach for realizing identity such as memory addresses, assigned numbers, or combinations of attribute values.* This strong notion of identity is an integral and important part of object orientation.

Objects and their relationships are delineated in an instance diagram, such as that in Figure 2.1. We denote an object by a box with an object name in the top portion of the box 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. An optional second portion of the box may list each attribute name followed by an equal sign and the value. We do not list operations because they do not vary among objects of the same class. In the figure, object *aBinaryTree* belongs to the class *BinaryTree* and has no specified attribute values. Object *IAH* is in class *Airport* and has the values *IAH*, *Intercontinental*, and *Central*.

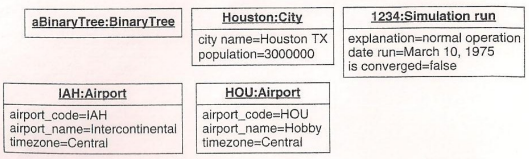


Figure 2.1 Objects

* If you use an object-oriented DBMS, it will impose an approach to identity and save you much work. In contrast, relational DBMSs and files provide less support for identity. Parts 2 and 3 show how to deal with identity for relational DBMSs and files.

When you construct models you must exercise judgment in deciding which objects to show and which objects to ignore. An object represents only the *relevant* aspects of a problem. It is not helpful to model extraneous detail; the scope of a model is driven by what is needed for an application.

Figure 2.1 demonstrates three conventions for separating portions of a name: spaces, underscores, and intervening capital letters. For a particular problem, you should consistently use one of these. From now on, we will use the convention of mixed case.

2.1.2 Classes

An object is an ***instance***—or occurrence—of a class. A ***class*** is a description of a group of objects with similar properties (object attributes), common behavior (operations and state diagrams), similar relationships to other objects, and common semantics. Classes provide a mechanism for sharing across similar objects. Classes often appear as common nouns and noun phrases in problem descriptions or in discussions with users. Some classes have real-world counterparts (person and company) while others are conceptual entities (simulation run and equation). Still other classes (binary tree and array) are purely artifacts of implementation. The choice of classes depends on the nature and scope of an application and is a matter of judgment.

Classes and their relationships are delineated in a class diagram, such as that in Figure 2.2, which shows the classes that correspond to the objects in Figure 2.1. We denote a class by a box with the class name in the top portion of the box. An optional second portion of the box may list attributes. An optional third portion of the box may list operations. Our convention is to list the class name in boldface, capitalize the first letter of a class name, and use lowercase letters for the first letter of attribute and operation names. We use singular nouns for the names of classes. We center the class name and left-align the attributes and operations.

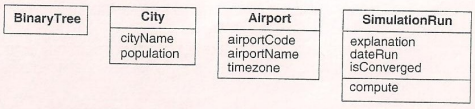


Figure 2.2 Classes

In Figure 2.2 class *BinaryTree* has no attributes and no operations specified. If *BinaryTree* were part of a large, multipage class diagram, it could have attributes and operations specified elsewhere, but they are not shown in Figure 2.2. Class *City* has attributes *cityName* and *population*. Class *Airport* has attributes *airportCode*, *airportName*, and *timezone*. We do not show operations for *City* and *Airport*. This does not imply a lack of operations; we just have not shown any. Class *SimulationRun* has attributes *explanation*, *dateRun*, and *isConverged* as well as operation *compute*.

Class diagrams and instance diagrams are the expression of the object model. As a class describes a group of objects, so too a class diagram describes a group of instance diagrams. Thus, a class diagram lets you abstract many individual instance diagrams to document data structure thoroughly. We will revisit class and instance diagrams after reviewing more object-modeling constructs.

The term "class" has been overloaded with several meanings in the literature. [Cattell-91] delineates three uses of the term "class."

- A class defines the ***intent***—the structure and behavior of objects of a particular type. The intent is the theme, essence, or semantic meaning of a class. For example, the intent for *SimulationRun* consists of the properties (*explanation*, *dateRun*, *isConverged*), behavior (*compute*), and relationships to other classes of objects (discussed in Section 2.2). Most object-oriented DBMSs and relational DBMSs adopt this definition of a class and it is the meaning we assign to "class" in the OMT methodology.
- A class defines a ***representative***—types represented by objects themselves. This meaning arises with the prototypical or delegation approach to object-oriented programming (such as the programming language *Self* [Ungar-87]). For example, you could define a new *SimulationRun* object by noting the deviations with respect to *SimulationRun 1234*.
- A class defines an ***extent***—the set of objects with a particular type. For example, the extent for *SimulationRun* is the set of all simulation run objects. In a relational database, the table combines the notions of intent and extent; a table name refers to both the structure of the table and the collection of instances stored in the table. Most object-oriented DBMSs do not automatically maintain the extent of a type.

2.1.3 Values and Object Attributes

A ***value*** is a piece of data. Values are found with enumerations and examples in problem documentation. An ***object attribute*** is a named property of a class that describes a value held by each object of the class. Object attributes are often discovered through adjectives or by abstracting typical values. The following analogy holds: Object is to class as value is to attribute. Object models are dominated by structural constructs, that is classes and relationships. Attributes are of lesser importance and serve to elaborate classes and relationships.

As the instance diagram in Figure 2.1 shows, values are listed in a second portion beneath the object name. The *IAH* object has values *IAH*, *Intercontinental*, and *Central*. Similarly, as the class diagram in Figure 2.2 shows, object attributes are listed in a second portion beneath the class name. The *Airport* class has attributes *airportCode*, *airportName*, and *timezone*.

Do not confuse values with objects. Objects have identity; values do not. Three persons with the name *Jim Smith* are three separate objects with their own identities. In contrast, all occurrences of the value "Jim Smith" (a string) are indistinguishable. Objects may be described with attributes and operations and participate in relationships; values merely exist.

Furthermore, a value should not refer to an object. The value of an attribute should be a simple primitive (string or number) or a collection of simple primitives. A reference to another object is properly modeled as a link, as we describe in Section 2.2.1.

When creating an analysis model, you should not include an attribute merely as an internal identifier for the instances of a class; object identifiers are implicit in analysis models. You should list only attributes that have meaning for the application. In contrast, during design you may show internal identifiers. For example, you may use internal identifiers to clarify the design of relational database tables.

In Figure 2.3, *airportID* is an inappropriate attribute for analysis because it has no intrinsic meaning. It merely refers to the identifier generated by an object-oriented DBMS or the sequence number that can be used to implement identity with a relational DBMS. In contrast, it can be quite proper to list *vehicleNumber* during analysis, because such a number can be found on the dashboard of a car.

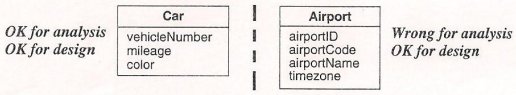


Figure 2.3 Do not show object identifiers in an analysis model

2.1.4 Operations and Methods

An ***operation*** is a function or procedure that may be applied to or by objects in a class. Each operation has a target object as an implicit argument. As Figure 2.2 shows, operations are listed in the third portion of the box. Thus, class *SimulationRun* has operation *compute* with an implicit target of a simulation run. Operation names may be followed by optional details such as an argument list and result type. Some operations are ***polymorphic***, that is applicable to many classes. A ***method*** is the implementation of an operation for a class.

For example, suppose class *Equipment* has an operation *calculateCost*. Operation *calculateCost* is polymorphic because there are different methods (formulas) for computing cost depending on the kind of equipment (tanks, pumps, conveyors, and so on). All these methods logically perform the same task but may be implemented by different pieces of code. Object-oriented software automatically chooses the appropriate method to use in computing an operation according to the class of the target object.

2.2 LINK AND ASSOCIATION CONCEPTS

2.2.1 Links and Associations

A ***link*** is a physical or conceptual connection between objects. Most links relate two objects, but some links relate three or more objects. An ***association*** is a description of a group of links with common structure and common semantics. Thus, a link is an instance of an association. The links of an association relate objects from the same classes and have similar properties (link attributes). An association describes a set of potential links in the same way a class describes a set of potential objects. Links and associations often appear as verbs in problem statements.

Figure 2.4 and Figure 2.5 illustrate notation for links and associations, respectively. A link is denoted by a line connecting related objects; a line may consist of several line segments. For example, both the IAH and HOU airports serve the city of Houston. An association connects related classes and is also denoted by a line (with possibly multiple line segments). For example, airports serve cities. (The two solid balls are multiplicity symbols, which we will discuss in the next section.) Our convention is to show link and association names in italics and to try to confine line segments to a rectilinear grid.

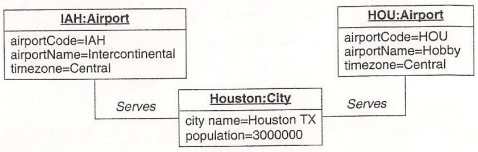


Figure 2.4 Links

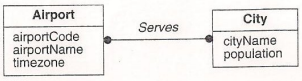


Figure 2.5 An association

The association name is optional, if the model is unambiguous. Ambiguity arises when there are multiple associations between the same classes (*person works for company* and *person owns stock in company*). When there are multiple associations, you must use association names or role names (Section 2.2.3) to resolve the ambiguity.

During analysis you may traverse links and associations in either direction, even though an association name may read in a single direction. For example, the association *an airport serves multiple cities* also implies that a city may be served by multiple airports.

Do not confuse links with objects. An object has inherent identity and exists in its own right. A link derives its identity from the related objects and cannot exist in isolation. A link can exist only if all objects to which it refers also exist.

Furthermore, do not confuse links with values. A value is a primitive such as a string or number. A link is a relationship between objects. During analysis, you should model all references to objects with links and recognize groups of similar links with associations. During design, you can choose to implement associations with pointers, foreign keys, or by some other means. Figure 2.6 illustrates two of several design models you could use to implement the *airport serves city* analysis model. You should delay the decision of how to implement associations until you complete analysis and understand a problem.

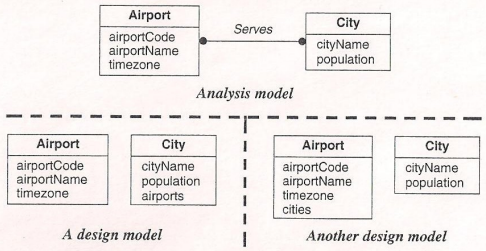


Figure 2.6 Use associations to model references to objects during analysis

Associations are an important and distinctive aspect of OMT modeling. Associations are taken for granted in the database community, where they have long been used with Entity-Relationship modeling. Most object-oriented DBMSs and the ODMG standard [Cattell-96] support simple associations. Most relational DBMSs provide referential integrity that also can support associations (see Chapter 14). However, associations have been overlooked by most object-oriented languages, as well as many object-oriented development methodologies. An association is not the same as a pointer. Proper use of associations will take some effort from those accustomed to programming and who may be less familiar with databases.

Figure 2.7 demonstrates the utility of associations with a simple model of information that might help a traveler choose a hotel. Each metropolitan area has a number of hotels, some of which are affiliated with a hotel chain. For example, the Hilton hotel in downtown Fort Wayne, Indiana belongs to the Hilton hotel chain. Most hotels accept a variety of credit

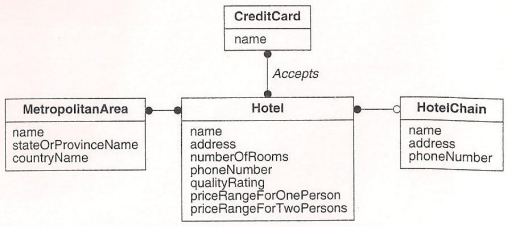


Figure 2.7 Associations in an object model for hotel selection

cards; the major credit cards are accepted by most hotels. (The solid and hollow balls are multiplicity symbols, which we will discuss in the next section.)

2.2.2 Multiplicity

Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. 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. Figure 2.8 summarizes multiplicity combinations. Solid balls denote "many" multiplicity, meaning zero or more. For example, a metropolitan area may have many hotels. A hollow ball denotes "zero or one" multiplicity. For example, a hotel may or may not belong to a hotel chain. The lack of a symbol at the end of an association line means exactly one. A driver's license belongs to a single person.

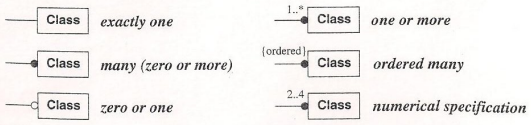


Figure 2.8 OMT notation for multiplicity

You may annotate a solid ball to indicate a more specific multiplicity, such as "1..*" or "2..4" multiplicity. You may also list several ranges such as "3..5,7..9,15..20" multiplicity. You may specify that the collection of objects for "many" multiplicity is ordered.

Figure 2.9 illustrates multiplicity with an excerpt of a model for airline flight reservations. A flight may be booked with multiple flight reservations. (Hopefully there will be numerous reservations or the plane will be empty!) The TW 250 flight from St. Louis to Milwaukee on October 23, 1995 is an example of a flight. The fare code for a flight reservation determines the fare that the airline will charge. For example, there are fare codes for first class, unrestricted coach, and weekend excursions. A trip reservation consists of multiple flight reservations that are ordered as a passenger proceeds from one city to the next. The airlines use a record locator to identify a trip reservation in their information systems. The passenger may note the record locator, especially for communicating with the airline about a complicated trip. A ticket may be issued for a trip, subject to the actual fare charged.

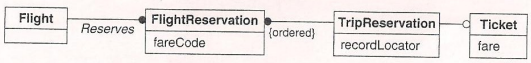


Figure 2.9 A simple object model for airline flight reservations

We do not show any attributes for *Flight* in Figure 2.9. A class need not have attributes to be included in an object model. A class may be significant because of its relationship to other classes, or a class may have attributes that have not been noted yet. We are going to elaborate the flight reservations example as we proceed in this chapter and will show attributes for *Flight* later.

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. As Figure 2.10 and Figure 2.11 show, if you want two links between the same objects, you must have two associations.†

Figure 2.10 A pair of objects can be instantiated at most once per association

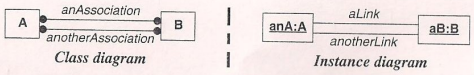


Figure 2.11 Use multiple associations to model multiple links between the same objects

† We thank Michael Chonoles for the suggestion to clarify this possible confusion.

Our notation for multiplicity follows that in [Rumbaugh-91], but you may use that in [UML-98] if you prefer. The notation in [UML-98] drops the solid and hollow balls and explicitly lists all multiplicities; "*" is a special symbol that denotes "many." Figure 2.12 restates Figure 2.9 to illustrate the UML notation. We believe the multiplicity notation in [Rumbaugh-91] is more readable than the UML multiplicity notation, especially for large models with many classes. Visual conciseness and brevity are important in presenting large models.

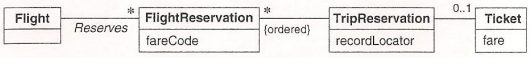


Figure 2.12 The flight reservations model using the multiplicity notation from [UML-98]

2.2.3 Roles

A ***role*** is one end of an association and may be assigned an explicit name. Role names often appear as nouns in problem descriptions. As Figure 2.13 shows, a role name is written next to the class-association intersection. When there is a single association between a pair of distinct classes (as in Figure 2.9), the names of the classes often serve as good role names, in which case you may omit the role names from the diagram.

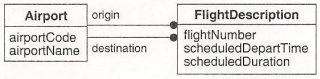


Figure 2.13 Roles in an object model for flight descriptions

FlightDescription in Figure 2.13 is different from *Flight* in Figure 2.9. A flight description refers to the published description of air travel between two airports. In contrast, a flight refers to the actual travel made by an airplane on a particular date. Thus, TW 250 is an example of a flight description; TW 250 on October 23, 1995 is an example of a flight.

Role names are especially convenient for traversing associations, because you can treat a role name as a pseudo attribute. In Figure 2.13 the notation *aFlightDescription.origin* refers to the airport at which a flight originates. (We use the convention of preceding a class name by "a" to refer to an object. Thus *aFlightDescription* refers to an object of the *FlightDescription* class.) Similarly, *aFlightDescription.destination* refers to the airport at which a flight terminates. Since a role is a pseudo attribute, the role name must not clash with any other attribute or role of the origination class. Thus, *FlightDescription* may not have any attributes or other target roles called *origin* or *destination*. Chapter 5 elaborates the notation for traversing object models.

Some modelers prefer to use directed association names, as Figure 2.14 shows. With directed association names you can more easily translate a model into natural language, but we

consider this a minor benefit. Besides, a developer should never disseminate a "naked" model; there should always be definitions of key terms and a natural-language narrative to guide readers through the model and document the rationale of various modeling decisions. We strongly favor role names because they can be easily referenced when traversing an object model, unlike directed relationship names (see Chapter 5).

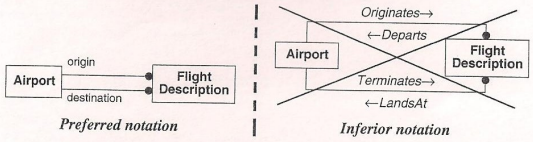


Figure 2.14 Use roles instead of directed association names

Role names are optional if a model is unambiguous. Ambiguity occurs when there are multiple associations between the same classes (Figure 2.13) or an association between objects of the same class (a ***reflexive association***, Figure 2.15). You can use association names or role names to resolve multiple associations between the same classes. You must use role names to clarify an association between objects of the same class.

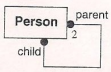


Figure 2.15 Roles can clarify an association between objects of the same class

Roles let you unify multiple references to the same class. When constructing class diagrams you should properly use roles and not introduce a separate class for each reference, as shown in Figure 2.16. In the wrong model, a person with a child is represented by two instances, one for the child role and one for the parent role. In the correct model, one person instance participates in two links, once as a parent and once as a child.

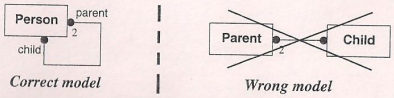


Figure 2.16 Use roles to model multiple references to the same class

2.2.4 Link Attributes

An object attribute is a named property of a class that describes a value held by each object of the class. Similarly, a ***link attribute*** is a named property of an association that describes a value held by each link of the association. A link attribute is a property of the association and should not be ascribed to an associated class. Link attributes are often discovered through adverbs in a problem statement or by abstracting known values.

A link attribute is denoted by a box attached to the association by a dashed line. The top portion of the box may be left blank or may contain the association name. One or more link attributes may appear in the second portion of the box. In Figure 2.17, a judge assigns a score to the efforts of a competitor (*Trial*) at an athletic meet. The *score* is a link attribute of the association between *Trial* and *Judge*.

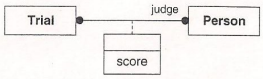


Figure 2.17 A link attribute is a property of the association

Many-to-many associations provide the most compelling rationale for link attributes. Such an attribute is unmistakably a property of the link and cannot be assigned to either object. In Figure 2.17, the score depends on both the trial and the judge. In contrast, for one-to-one and one-to-many associations, it is easier to confuse link attributes with object attributes. You will avoid errors in your applications if you construct your analysis models by directly describing objects and links. As a model evolves toward design and implementation, you can always restructure for more efficient execution.

2.2.5 Association Classes

An ***association class*** is an association whose links can participate in subsequent associations. An association class has characteristics of both an association and 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 participate in associations.

An association class is denoted by a box attached to an association by a dashed line—the same notation as for a link attribute. This correspondence in notation is appropriate because a link attribute is really just a simple association class. Like an ordinary class, an association class may have attributes and operations. Figure 2.18 shows alternative models for authorization in a relational DBMS.

A user may own multiple tables. The owner of a table may authorize one or more other users to access the table; an authorized user may grant further permissions. For example, user *A* may own a table and authorize users *B* and *C* to access the table; for these associations user *A* is the *grantor* and users *B* and *C* are *grantees*. User *B* may, in turn, authorize user *D* to access the table.

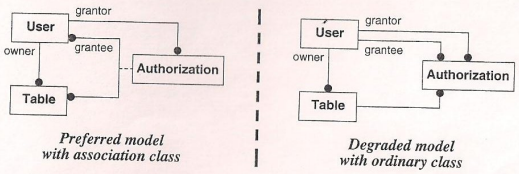


Figure 2.18 An association class lets you specify identity and navigation paths precisely

Association classes are an important facet of OMT modeling because they let you specify identity and navigation paths precisely. In Figure 2.18, the model with the association class clearly specifies that a table and a grantee imply an authorization and that a grantor may grant many such authorizations. The association class introduces a meaningful asymmetry to the model. In contrast, the degraded model in Figure 2.18 is symmetric and the normal path for traversing the model is not obvious. The degraded model has promoted authorization to a class with its own identity and has suppressed the fact that grantee and table directly relate.

In Figure 2.17, *Trial* is really an association class that describes a person competing in an athletic event. Figure 2.19 improves the model by using an association class.

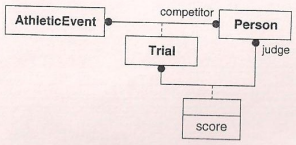


Figure 2.19 An association class for an athletic meet

2.2.6 Qualified Associations

A ***qualified association*** is an association in which the objects in a "many" role are partially or fully disambiguated by an attribute called the ***qualifier***. One-to-many and many-to-many associations may be qualified. A qualifier selects among the target objects, reducing the effective multiplicity, often 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. As we will describe in Part 3, you can use a DBMS index to enforce a qualified association with a target multiplicity of "one."

Figure 2.20 illustrates the most common use of a qualifier—to yield a single target object. A bank has many accounts, with the combination of a bank and account number yielding a single account. Without a qualifier many objects are referenced; with the qualifier, each reference is to a specific object. In the "qualified" model, the second portion of the *Account* box is empty; this is because we have chosen not to list any attributes for *Account*.

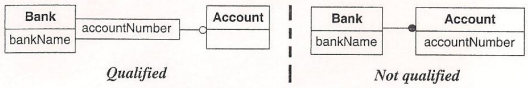


Figure 2.20 Qualification increases the precision of a model

Both models are acceptable, but the qualified model adds information. The qualified model adds a multiplicity constraint, that the combination of a bank and account number yields one account. The qualified model is actually more correct, because an account number really has no meaning unless the bank is also specified. The qualified model conveys the significance of *account number* in traversing the model, as methods will reflect. Chapter 3 presents more complex examples of qualified associations.

Figure 2.21 shows another example of a qualifier. A flight description pertains to many flights; the combination of a flight description and a departure date specifies a flight.

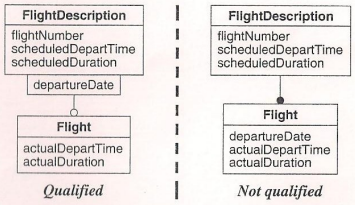


Figure 2.21 A simple qualified association for airline flights

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 2.21, for example, *FlightDescription* plus *departureDate* yields *Flight*.

Qualification frequently arises through the use of names. Few names are globally unique; most names have meaning within some context. For example, an account number has meaning only when combined with the bank holding the account. Qualification can also

arise through explicit ordering. For example, if the columns in a table are ordered, a table plus a column number yields a column.

2.3 GENERALIZATION CONCEPTS

2.3.1 Generalization

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, state diagrams, and associations; the subclasses add specific attributes, operations, state diagrams, and associations. There can be multiple levels of generalization relationships. An instance of a subclass is simultaneously an instance (transitively) of all its superclasses.

Specialization provides another perspective on a system's structure. Specialization has the same meaning as generalization but takes a top-down perspective, starting with the superclass and splitting out variations. In contrast, generalization takes a bottom-up view, starting with the subclasses and abstracting commonality.

Simple generalization organizes classes into a hierarchy; each subclass has a single immediate superclass. (Chapter 3 discusses a more complex form of generalization relationship in which a subclass may have multiple immediate superclasses.) Figure 2.22 shows several kinds of financial instruments that could be managed within an investment portfolio. All financial instruments have a name and a current value that is expressed in a currency. Stocks have a quarterly dividend that is set by the board of directors. Bonds have a maturity date, maturity value, and either a fixed interest rate or a variable interest rate that is computed from a reference rate. There are various types of insurance, such as disability, life, and hazard. An annual payment may be required to maintain insurance coverage.

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. You may also show a discriminator next to a subclass tree.

The ***discriminator*** is an attribute that has one value for each subclass; the value indicates which subclass further describes an object. The discriminator is simply a name for the basis of generalization. Discriminators are superfluous for object-oriented DBMSs, but they can be useful for implementations that divide objects into multiple records. For example, with a relational DBMS you might implement each class with a separate table and store a record at each generalization level of an object. In Figure 2.22 *financialInstrumentType* and *bondType* are discriminators.

Generalization is an important construct for both implementation and conceptual modeling. Generalization facilitates type checking and reuse in object-oriented languages. Much of the literature emphasizes these programming uses of generalization. However, generalization has importance that transcends implementation. Generalization is also significant in the early stages of software engineering because it lets you call attention to patterns of similarities and differences. Often there are different ways to group classes; with generalization

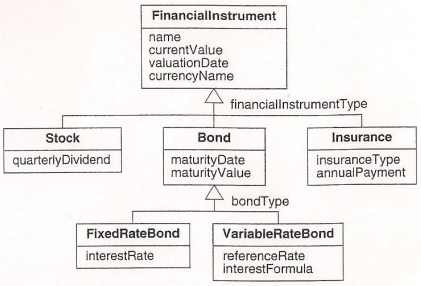


Figure 2.22 Generalization organizes classes by their similarities and differences

you make a statement about the relative importance of these different organizational bases. The improved depth of understanding leads to systems that are better conceived and better architected. The object-modeling fabric of classes, associations, and generalizations provides a rich, but concise, medium for thinking about applications.

2.3.2 Inheritance

Generalization is the structural relationship that permits the ***inheritance*** mechanism to occur. A subclass ***inherits*** the attributes, operations, state diagrams, and associations of its superclasses. Inherited properties can be reused from a superclass or overridden in the subclasses; new properties can be added to the subclasses. For example, in Figure 2.22 an object in the *FixedRateBond* class has seven attributes: *name*, *currentValue*, *valuationDate*, *currencyName*, *maturityDate*, *maturityValue*, and *interestRate*.

Simple generalization is really another term for ***single inheritance***, and the case in which a subclass may have multiple immediate superclasses is called ***multiple inheritance***. Chapter 3 describes multiple inheritance in more detail.

2.4 SAMPLE DIAGRAMS

2.4.1 Class Diagram

A ***class diagram*** is a graphical representation that describes objects and their relationships. Figure 2.23 presents a class diagram for an airline flight reservation system. We used portions of this diagram in earlier examples.

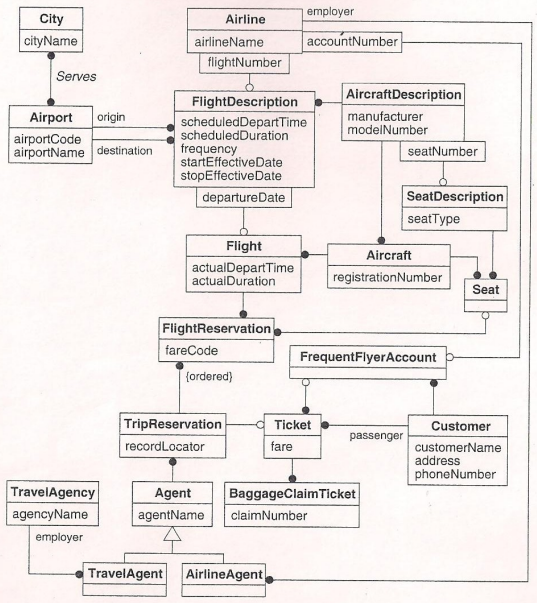


Figure 2.23 Class diagram for an airline flight reservation system

An airport serves many cities, and a city may have multiple airports. Airlines operate flights between airports. A flight description refers to the published description of air travel between two airports. In contrast, a flight refers to the actual travel made by an airplane on a particular date. The frequency indicates the days of the week for which the flight description applies. The start and stop effective dates bracket the time period for which the published flight description is in effect.

A sample flight description is TW 250 from St. Louis to Milwaukee. For the fall of 1995, TW 250 is scheduled to depart at 7:42 AM, has a scheduled duration of one hour eighteen minutes, is scheduled to be flown by a DC9 plane, and operates every day of the week except Sunday. A sample flight is TW 250 on October 23, 1995, which actually departed at about 7:45 AM and had an actual duration of one hour five minutes.

Each flight is normally booked by multiple reservations at various fare codes. The fare code governs the fare that is charged. Fare codes reflect multiple factors, such as the number of days of advance purchase, refundability, whether the passenger is staying over a Saturday night, and whether the seat is coach or first class (*seatType*). A passenger may be assigned a seat for a flight reservation. A seat may have multiple flight reservations, but hopefully no more than one for a flight!

A trip reservation consists of a sequence of flight reservations. For example, we could make a trip reservation for traveling from St. Louis to Milwaukee on TW 663 on October 20 and returning to St. Louis on TW 250 on October 23. The airlines use record locators to find a particular trip reservation quickly and unambiguously. Experienced passengers will often use these record locators for communicating with the airline. Trips are reserved by agents, who either work for an airline or a travel agency. A ticket may be issued for a passenger for a trip reservation, and a frequent flyer account may be noted.

We chose not to combine the *Customer* and *Agent* classes. In one sense, we were tempted to combine them; *Customer* and *Agent* could be regarded as roles of *Person*. However, there seems to be little commonality between agents and customers within a flight reservation system. Furthermore, we would like to allow some unusual customers, such as corporations, possibly animals, and inanimate objects. For example, a corporation might purchase an airplane seat to transport some delicate equipment.

For ease of layout, you may repeat class names within a class diagram, especially for large, multipage diagrams. However, each occurrence of a class name must refer to the same thing. For example, you cannot use *table* within the same diagram to refer to a relational database table and a piece of furniture. Attribute names (including roles as pseudo attributes) and method names may be globally unique, or unique within a class or association.

You should try to avoid intersecting association lines in your diagrams. For example, the right diagram in Figure 2.24 has three possible meanings: associations *AC* and *BD*, *AB* and *CD*, or *AD* and *BC*. Often you can adjust the layout of a diagram so there is no need for intersecting lines, as we did in Figure 2.23. Otherwise you can use a crossing symbol to avoid ambiguity.

By now we have presented enough object-modeling constructs, so that you should be able to read object diagrams. However, we have not yet shown you a process for constructing diagrams. This is not a problem in Part 1, because we have chosen problems with small models. However, for large problems you will need a systematic approach to constructing diagrams, and we present such a process in Chapter 8.

2.4.2 Instance Diagram

An ***instance diagram*** is a diagram that involves only objects, links, and values. Instance diagrams are useful for debugging and especially for understanding class diagrams.

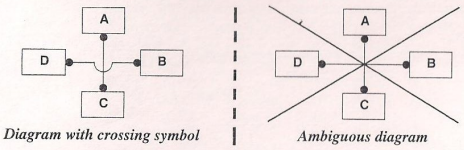


Diagram with crossing symbol *Ambiguous diagram*

Figure 2.24 Intersecting association lines can be ambiguous

Often there are alternative ways to conceive a model, and it is not obvious which model is best. Instance diagrams can help you understand the capabilities and shortcomings of different representations. The most appropriate model then depends on the requirements of an application. As an example, consider the two class diagrams for directed graphs in Figure 2.25. (Figure E2.7 and Chapter 4 have additional models for directed graphs.)

A directed graph consists of a set of nodes and a set of edges. Each edge connects two nodes and has an arrow indicating the direction of flow. The left model describes directed graphs with at most one edge between nodes; edges are implicitly represented by the *parent-child* association between *Branch* and *Node*. By definition each node has one or more parents except root nodes. The right model is more powerful and can describe any directed graph.

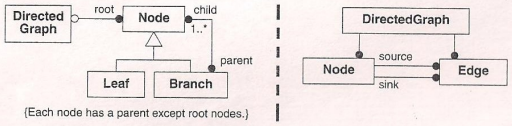


Figure 2.25 Class diagrams for directed graphs

Figure 2.26 shows a sample directed graph and the instance diagrams for the respective class diagrams. To reduce clutter we omit the links to the *DirectedGraph* object for the instance diagram for the right model. Note that the instance diagram for the left model directly corresponds to the graph. The graph consists of nodes that are connected by edges; similarly, the instance diagram consists of objects that are connected by links. In contrast, the instance diagram for the right model is more complex because it elevates both nodes and edges to the status of objects.

Figure 2.27 shows another directed graph. Again we omit the links to *DirectedGraph* for the instance diagram for the right model to reduce clutter. Such a graph cannot be represented by the left model; the left model cannot represent multiple edges between nodes.

Our point in presenting these diagrams is that both models are correct. If an application involves directed graphs with at most one edge between nodes, the left model is preferable

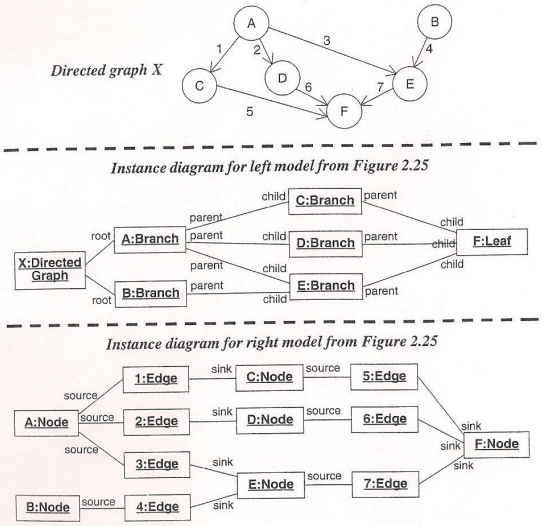


Figure 2.26 Instance diagrams for a directed graph

because it is simpler and structurally enforces the constraint that there not be multiple edges between nodes. On the other hand, the right model is more appropriate for applications that involve directed graphs in their more general form.

2.5 PRACTICAL TIPS

We have gleaned from our experience the following tips for constructing object models.

- **Scope of model.** Make sure you clearly understand the problem to be solved. The content of an object model is driven by relevance to an application. (Section 2.1.1)

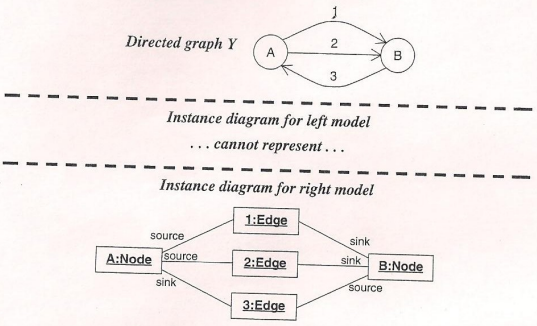


Figure 2.27 Instance diagrams for a directed graph

- **Simplicity**. Strive to keep your models simple. A simple model is easier to understand. 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. (Section 2.4.2)
- **Layout of diagram**. Draw your class diagrams in a manner that elicits symmetry. Often there is a superstructure to a problem that lies outside the OMT notation that we have presented. Try to position important classes so that they are visually prominent on a diagram. Try to avoid crossing lines on diagrams.
- **Names**. Carefully choose names. Names carry powerful connotations. Names should be descriptive, crisp, and unambiguous. Choosing good names is a difficult aspect of object modeling. Persons often argue over names, but the arguments are worthwhile because they cause users and developers to think more deeply about a model and reach a consensus. You should use singular nouns for the names of classes.
- **Implementation of associations**. During analysis use associations to capture the true intent. During design you can choose to implement associations with pointers, foreign keys, or by some other means. (Section 2.2.1)
- **Multiplicity**. You should carefully review multiplicity decisions in your object models. Challenge roles with a multiplicity of exactly one. Often the object is optional and "zero-or-one" multiplicity is more appropriate. Sometimes "many" multiplicity may be needed. (Section 2.2.2)

- **Roles**. Be alert for multiple uses of the same class. Use roles to unify references to the same class. (Section 2.2.3)
- **Link attributes**. During analysis, do not collapse link attributes 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 2.2.4)
- **Qualifiers**. Challenge roles with a multiplicity of "many." A qualifier can often improve the precision of an association and highlight important navigation paths. (Section 2.2.6)
- **Review of models**. Try to get others to review your models. Expect that your models will require revision. Object 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.
- **Instance diagrams**. Selectively use instance diagrams to explain subtleties of a class diagram. However, do not feel compelled to show a large number of instance diagrams, because they can be voluminous and are tedious to construct and read. (Section 2.4.2)

2.6 CHAPTER SUMMARY

The object model describes the data structure of a system, that is objects and their relationships. Object models are built from classes that relate to each other through associations and generalizations. An object is a concept, abstraction, or thing that can be individually identified and has meaning for an application. A class is a description of a group of objects with similar properties (object attributes), common behavior (operations and state diagrams), similar relationships to other objects, and common semantics.

A link is a physical or conceptual connection between objects and is an instance of an association. An association is a description of a group of links with common structure and common semantics. An association describes a set of potential links in the same way that a class describes a set of potential objects. Associations are an important and distinctive aspect of OMT modeling that most object-oriented DBMSs and relational DBMSs support.

Multiplicity specifies the number of instances of one class that may relate to a single instance of an associated class. A role is one end of an association. Each role may have an explicit name and a multiplicity. A link attribute is a named property of an association that describes a value held by each link of the association. An association class is an association whose links can participate in subsequent associations. A qualified association is an association in which the objects in a "many" role 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."

Generalization is the relationship between a class (called the superclass) and one or more variations of the class (called the subclasses). Generalization organizes classes by their similarities and differences, structuring the description of objects. Generalization is the

structural relationship that permits the inheritance mechanism to occur. A subclass inherits the attributes, operations, state diagrams, and associations of its superclasses. Inherited properties can be reused from a superclass or overridden in the subclasses; new properties can be added to the subclasses.

This chapter has presented basic object modeling concepts—the concepts that frequently occur and are sufficient for modeling many applications. Figure 2.28 lists the key concepts for this chapter.

association	identity	multiplicity	qualifier
association class	inheritance	object	role
attribute	instance	object attribute	specialization
class	instance diagram	object model	subclass
class diagram	link	operation	superclass
discriminator	link attribute	polymorphism	value
generalization	method	qualified association	

Figure 2.28 Key concepts for Chapter 2

CHANGES TO OMT NOTATION

We have made the following changes to the notation in [Rumbaugh-91] for the sake of compatibility with the UML notation [UML-98]:

- **Object**. We changed the notation for an object to a box. The object name, a colon, and the class name in the top portion of the box are underlined.
- **Multiplicity**. For the situation where multiplicity is "many" and more restrictive than "0..*", we *overload* the solid ball with a textual specification. We show *both* the solid ball and the specification. This was not made clear in [Rumbaugh-91].
- **Link attribute**. A link attribute is denoted by a box attached to the association by a dashed line. A link attribute is a special case of an association class.
- **Association class**. An association class is also denoted by a box attached to the association by a dashed line.
- **Generalization** 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. (This is our choice of style and is allowed by the UML.) You may show an optional discriminator next to a subclass tree.

We have made one further change to [Rumbaugh-91] not mentioned by [UML-98]:

- **Crossing symbol**. We use a crossing symbol to avoid ambiguity when lines in diagrams must cross.

BIBLIOGRAPHIC NOTES

Our presentation in this chapter is similar to that in [Rumbaugh-91]. Overall, we have seen little need to revise the object model. The current notation works well in practice and strikes a reasonable compromise between expressiveness and simplicity.

[Loomis-87] is the first publication about the OMT object-modeling notation. Loomis articulated the database viewpoint and Rumbaugh articulated the programming viewpoint in developing the initial object-modeling notation.

Pages 112–113 of [Wirfs-Brock-90] explain the proper use of inheritance in terms of Venn diagrams.The responsibilities of a subclass should completely encompass those of its superclasses. (Wirfs-Brock responsibilities are similar to OMT attributes and methods.) Subclass responsibilities cannot partially overlap superclass responsibilities; if this were to occur, then the subclass would not provide some superclass behavior, thus violating the meaning of inheritance. With inheritance every instance of a subclass must be an instance of all its superclasses.

The information model of [Shlaer-88] is similar to the OMT object model. Both notations emphasize structure and have been heavily influenced by database design techniques. A major difference is that the OMT object model applies to both analysis and design. During analysis OMT models are constructed in the conceptual realm; during design OMT models are elaborated and optimized for implementation needs. In contrast, the information model of [Shlaer-88] is built about relational database concepts. The heavy influence of relational database concepts is distracting during analysis, especially when a relational database is not the ultimate implementation medium.

REFERENCES

[Cattell-91] RGG Cattell. *Object Data Management: Object-Oriented and Extended Relational Database Systems*. Reading, Massachusetts: Addison-Wesley, 1991.

[Cattell-96] R Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. San Mateo, California: Morgan Kaufmann, 1996.

[Khoshafian-86] SN Khoshafian and GP Copeland. Object identity. *OOPSLA'86 as ACM SIGPLAN 21*, 11 (November 1986), 406–416.

[Loomis-87] Mary ES 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.

[Rumbaugh-91] J Rumbaugh, M Blaha, W Premerlani, F Eddy, and W Lorensen. *Object-Oriented Modeling and Design*. Englewood Cliffs, New Jersey: Prentice Hall, 1991.

[Shlaer-88] Sally Shlaer and Stephen J Mellor. *Object-Oriented Systems Analysis: Modeling the World in Data*. Englewood Cliffs, New Jersey: Yourdon Press, 1988.

[UML-98] The following books are planned for the Unified Modeling Language:
Grady Booch, James Rumbaugh, and Ivar Jacobson. *UML User's Guide*. Reading, Massachusetts: Addison-Wesley.
James Rumbaugh, Ivar Jacobson, and Grady Booch. *UML Reference Manual*. Reading, Massachusetts: Addison-Wesley.

Ivar Jacobson, Grady Booch, and James Rumbaugh. *UML Process Book*. Reading, Massachusetts: Addison-Wesley.

[Ungar-87] David Ungar and Randall B Smith. SELF: The power of simplicity. *OOPSLA'87 as ACM SIGPLAN* 22, 12 (December 1987), 227–241.

[Wirfs-Brock-90] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Englewood Cliffs, New Jersey: Prentice Hall, 1990.

EXERCISES

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

2.1 (2) Explain the object model in Figure E2.1.

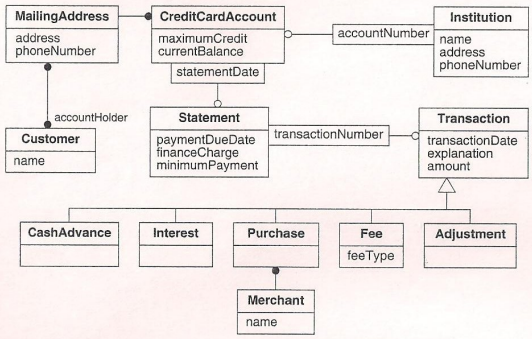


Figure E2.1 Object model for managing credit card accounts

2.2 (5) Add the following information to the object model of Figure 2.7:

a. A hotel may have a number of amenities, such as restaurant, lounge, fitness center, indoor pool, outdoor pool, golf, and tennis. Extend the model so that you can easily obtain the set of amenities for a given hotel and the set of hotels that offer one or more amenities.

b. Many hotels have variable pricing throughout the year; rates are low during off seasons and high during seasons of high demand. Extend the model so that it organizes pricing information by season.

2.3 (3) Construct an instance diagram for both models in Figure 2.18 with the following data. User *A* may own a table and authorize users *B* and *C* to access the table; user *B* may, in turn, authorize user *D* to access the table.

2.4 (4) Compare the object models in Figure E2.2. 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.

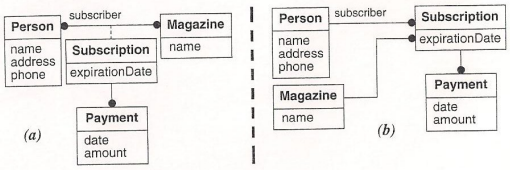


Figure E2.2 Object models for magazine subscriptions

2.5 (2) Refine the object model of Figure E2.3 by using generalization. The object model describes bibliographic citations such as those that appear in this book. A citation code is an abbreviation for referring to a paper or book. Position number refers to the ordering of the authors of a cited work.

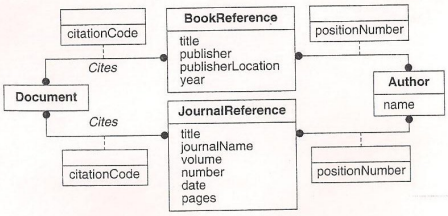


Figure E2.3 Object model for citations in a document

2.6 (5) Further improve the object model in Figure E2.3 through the use of qualifiers.

2.7 (4) Prepare an object model for personal automotive repair records. Include the following: date of repair, vehicle repaired, vehicle mileage, repair tasks, repair facility, cost, and comments. You should indicate tasks that fulfill warranty provisions. Hint: You should have the following classes: *Vehicle*, *RepairEvent*, *RepairTask*, and *VehicleRepairFacility*.

2.8 (7) Figure E2.4 shows a receipt for the purchase of a few items from a store. Prepare an object model that describes this information.

```
Acme food store
(314) 555-1234
198 State Street
Chesterfield, MO 63017

Description     Quantity     Price     Discount     Amount
-----------     --------     -----     --------     ------
milk                2         3.50        0.00        7.00
beans               5          .75        0.00        3.75
motor oil           1         2.00        0.00        2.00

Sub-total                    12.75
Tax                           0.76
Total                        13.51
Amount on credit card        13.51
Change                        0.00

Customer name          Brian Smith
Credit card number     1234 5678 1234 5678
Expiration date        05/99
Approval number        AP654321

Customer signature

Receipt number         345127
Date                   15 Sep 1996
Time                   16:43
Employee               14
Register               8
```

Figure E2.4 A receipt for a purchase from a store

2.9 (6) Figure E2.5 is an excerpt of the certificate of title form for a motor vehicle. Prepare an object model that describes this information. Represent the title number as a qualifier.

2.10 (7) Figure E2.6 shows a typical frequent flyer account summary. Prepare an object model that describes this information.


```
Title number --- TE125023

Vehicle identification Number - KN4MW6335L1226731
Year - 89                      Make - Mazda 323 car
Cylinders - 6                  Horsepower - 30
Previous state - NY            Mileage at time of transfer - 61515
Purchase date - 15 Mar 1992
Date issued - 10 Aug 1994

Owner - John and Mary Doe
        1561 Olive Street
        Scotia, NY 12309

First lien - First Bank
             1234 Main Street
             Schenectady, NY 12301

Second lien -
```

Figure E2.5 A simplified certificate of title for a motor vehicle

```
Frequent flyer account number: 123456789
Member since 06 Jun 1991

James Doe
1937 First Street
Creve Coeur, MO 63146

Activity      Prior      Credits    Remaining    Current    Current
as of:        balance    used       balance      Mileage    balance
-----------   -------    -------    ---------    -------    -------
17 Nov 1995   117,169          0     117,169        3,882    121,051

Date of   Airline   Flight   Service   Activity    Actual    Bonus
Travel    Partner   Number   Class                 Mileage   Mileage
-------   -------   ------   -------   --------    -------   -------
20 Oct      TW       0663      Y       STL-MKE     1,000     600
23 Oct      TW       0250      Y       MKE-STL     1,000     600
21 Aug      MC                         hotel                 500
17 Aug      SP                         phone                 182
```

Figure E2.6 A typical frequent flyer account summary

2.11 (6) Prepare an object model for the program listings in a television guide. For movies you should include the network, date, time, movie name, description, quality rating, and release date. For other programs you should include the network, date, time, series name, episode name, and description.

2.12 (4) Construct an object model for movie information. Include the following: actor, actress, category, date, director, length, movie, producer, rating, synopsis, title, and writer.

2.13 (8) Discuss the relative merits of the object models for a directed graph that are shown in Figure E2.7. This exercise highlights the issue of alternative structures for an object model. A directed graph consists of a set of nodes and a set of edges. Each edge connects two nodes and has an arrow indicating direction of flow.

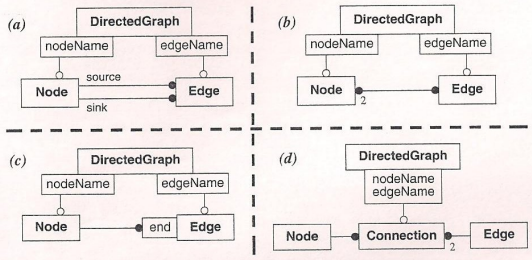


Figure E2.7 Alternative object models for a directed graph

2.14 (6) Construct instance diagrams for each of the four models in Figure E2.7 given the directed graph in Figure 2.27.

2.15 (7) Prepare an object model for checking accounts. Include the following classes: *Bank*, *CheckingAccount*, *Deposit*, *Check*, *InterestPayment*, *Fee*, *AccountOwner*, *Item*, and *Statement*.

A bank issues a checking account and assigns it a number. A checking account may have several account owners. An account owner may have multiple checking accounts. Deposits, checks, interest payments, and fees are all items that affect the balance for a checking account. There are different kinds of fees, including check printing fee, overdraft fee, and wire transfer fee. The bank issues periodic statements to the owners so that they can accurately monitor an account.

2.16 (7) Construct an object model for genealogical information. Include the following: names, sex, birth date, birth place, parents, marriages, children, death date, cause of death, and cemetery.

2.17 (7) Extend the genealogical object model to permit polygamy for males and females.

2.18 (7) Show two variations of the genealogical object model. One object model disallows remarriage between the same persons (as would occur with marriage, divorce, followed by marriage). The other object model permits remarriage between the same persons.

2.19 (6) Prepare an object 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.

2.20 (4) Prepare an instance diagram for the definition for *been* using your object model from the previous exercise.

3

Advanced Object Modeling

This chapter explains the advanced aspects of object modeling that you will need to model complex and large applications. It builds on the basic concepts in Chapter 2, so you should master the material there before reading this chapter. You can skip this chapter if you are just interested in getting a general understanding of object modeling.

3.1 OBJECT AND CLASS CONCEPTS

3.1.1 Instantiation

Instantiation is the relationship between an object and its class. The notation for instantiation is a dashed line from the instance to the class with an arrow pointing to the class; the dashed line is labeled with the legend *instance* enclosed by guillemets («»). Figure 3.1 shows this notation for *City* and its two instances *Bombay* and *Prague*. Making the instantiation relationship between classes and instances explicit in this way can be helpful in modeling complex problems and in giving examples.

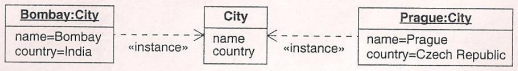


Figure 3.1 Instantiation relationships

3.1.2 Class Attributes and Operations

A ***class attribute*** is an attribute whose value is common to a group of objects in a class rather than peculiar to each instance. Class attributes can be used to store default or summary data

for objects. A ***class operation*** is an operation on a class rather than on instances of the class. The most common kind of class operations are operations to create new class instances. You can denote class attributes and class operations with an underline. Our convention is to list them at the top of the attribute box and operation box, respectively.

In most applications class attributes can lead to an inferior model. We discourage the use of class attributes. Often you can improve your model by explicitly modeling groups and specifying scope. For example, the upper model in Figure 3.2 shows class attributes for 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.

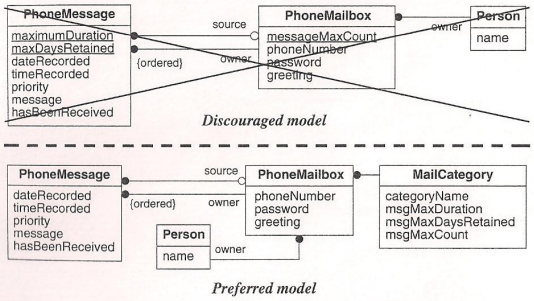


Figure 3.2 Instead of using class attributes, model groups explicitly

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

3.1.3 Attribute Multiplicity

Attribute multiplicity specifies the possible number of values for an attribute and is listed in brackets after the attribute name. You may specify a mandatory single value *[1]*, an optional single value *[0..1]*, an unbounded collection with a lower limit *[lowerLimit..*]*, or a collec-

tion with fixed limits *[lowerLimit..upperLimit]*. A lower limit of zero allows null values; a lower limit of one or more forbids null values. (***Null*** is a special value denoting that an attribute value is *unknown* or *not applicable*. See Chapter 9.) If you omit attribute multiplicity, an attribute is assumed to be single valued with nullability unspecified (*[0..1]* or *[1]*). In Figure 3.3 a person has one name, one or more addresses, zero or more phone numbers, and one birth date. Attribute multiplicity is similar to multiplicity for associations.

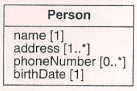


Figure 3.3 Attribute multiplicity

3.1.4 Candidate Keys for Classes

A ***candidate key*** for a class is a combination of one or more attributes that uniquely identifies objects within a class. (Section 3.2.3 discusses candidate keys for associations.) The collection of attributes in a candidate key must be minimal; no attribute can be discarded from the candidate key without destroying uniqueness. No attribute in a candidate key can be null. A given attribute may participate in multiple candidate keys.

For example, in Figure 3.4 *airportCode* and *airportName* are two candidate keys for *Airport*. The model specifies that each *airportCode* (such as IAH, HOU, STL, ALB) uniquely identifies an airport. Each *airportName* (such as Houston Intercontinental, Houston Hobby, Lambert St. Louis airport, and Albany NY airport) also uniquely identifies an airport.

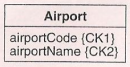


Figure 3.4 Candidate keys for a class

We indicate a candidate key for a class with the notation *CKn* in braces next to the appropriate attributes. The *n* is a number that differentiates multiple candidate keys. For a multi-attribute candidate key, multiple attributes have the *CKn* designation with the same value of *n*.

Some readers may recognize the term "candidate key" from the database literature, but the notion of a candidate key is a logical construct, not an implementation construct. It is often helpful to be able to specify the constraint that one or more attributes taken together are unique. Relational database managers and most object-oriented database managers can readily enforce candidate keys.

3.1.5 Domains

A ***domain*** is the named set of possible values for an attribute. The notion of a domain is a fundamental concept in relational DBMS theory, but really has broader applicability as a modeling concept. As Figure 3.5 shows, an attribute name may be followed by a domain and default value. The domain is preceded by a colon; the default value is preceded by an equal sign. Some domains are infinite, such as the set of integers; others are finite. You can define a domain intensionally (by formula), extensionally (by explicitly listing occurrences), or in terms of another domain.

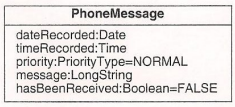


Figure 3.5 Assign a domain to an attribute rather than directly assign a data type

An ***enumeration domain*** is a domain that has a finite set of values. The values are often important to users, and you should carefully document them for your object models. For example, you would most likely implement *priorityType* in Figure 3.5 as an enumeration with values that could include *normal*, *urgent*, and *informational*.

A ***structured domain*** is a domain with important internal detail. You can use indentation to show the structure of domains at an arbitrary number of levels. Figure 3.6 shows two attributes that have a structured domain. An address consists of a street, city, state, mail code, and country. A birth date has a year, month, and day.

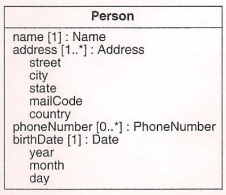


Figure 3.6 Structured domains

During analysis you can ignore simple domains, but you should note enumerations and structured domains. During design you should elaborate your object model by assigning a

domain to each attribute. During implementation you can then bind each domain to a data type and length.

Domains provide several benefits:

- **Consistent assignment of data types**. You can help ensure that attributes have uniform data types by first binding attributes to domains and then binding domains to data types.
- **Fewer decisions**. Because domains standardize the choices of data type and length, there are fewer implementation decisions.
- **Extensibility**. It is easier to change data types when they are not directly assigned.
- **Check on validity of operations**. Finally, you can use the semantic information in domains to check the appropriateness of certain operations. For example, it may not make sense to compare a name to an address.

Do not confuse a domain with a class. Figure 3.7 summarizes the differences between domains and classes. The objects of a class have identity, may be described by attributes, and may have rich operations. Classes may also be related by associations. In contrast, the values of a domain lack identity. For example, there can be many *Jim Smith* objects, but the value *normal* has only one occurrence. Most domain values have limited operations and are not described by attributes. During analysis we distinguish between domains and classes according to their semantic intent, even though some domains may be implemented as classes.

Classes	Domains
• A class describes objects.	• A domain describes values.
• Objects have identity.	• Values have no identity.
• Objects may be described by attributes.	• Most values are not described by attributes.
• Objects may have rich operations.	• Most values have limited operations.
• Classes may be related by associations.	• Domains do not have associations.

Figure 3.7 Classes and domains differ according to semantic intent

Do not confuse an enumeration domain with generalization. You should introduce generalization only when at least one subclass has significant attributes, operations, or associations that do not apply to the superclass. Do not introduce a generalization just because you have found an enumeration domain.

3.1.6 Secondary Aspects of Data

Occasionally you will encounter secondary aspects of attributes and classes that the OMT notation does not explicitly address [Blaha-93a]. This secondary data provides relevant information, but exists in a realm apart from the essence of an application. It is important to record secondary information without obscuring the focus of an application.

There are several kinds of secondary data for attribute values. Many scientific applications involve units of measure, such as inches, meters, seconds, and joules. Units of measure provide a context for values and imply conversion rules. For some numerical attributes you

must specify accuracy—whether the values are exact, approximate, or have some standard deviation. You may wish to note the source of data—whether the values are obtained from persons, the literature, calculations, estimates, or some other source. You may require the time of the last update for each attribute value.

Figure 3.8 illustrates secondary data for a financial instrument. Because the value of a financial instrument is stated for some date and currency, *valuationDate* and *currencyName* are secondary data. This is one design approach for dealing with secondary data for attributes. Chapter 9 presents additional approaches.

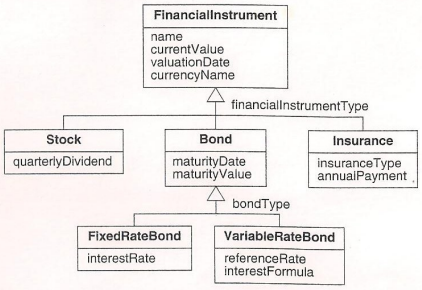


Figure 3.8 Secondary data for attribute values

Secondary data may also arise through policies and defaults that broadly apply to objects of various classes. Objects may have to be approved, logged, audited, distributed, and secured. They may also require multiple versions and may be persistent. For example, managers may need to approve critical data for some applications—the more important the data, the higher the level of approval. Permanent data must be logged to avoid accidental loss; transient objects may not be logged to speed processing. Updates to objects may necessitate an audit trail to protect against accidental and malicious damage. Some objects can be distributed over a network, while other objects may be limited to a single location. Objects may vary in their security level, such as none, unclassified, classified, and top secret. Some objects, such as alternative objects for an engineering design, may require versions. Other objects such as manufacturing records are not hypothetical and may not involve versions. Some objects may be persistent and require entry in the database, while other objects may be transient and need not exist beyond the confines of computer memory.

We have chosen not to augment the OMT notation for secondary data; too many variations are only occasionally required. We often use naming conventions to convey secondary

data. Naming conventions are simple, orthogonal to notation, and enrich a model. The drawback is that naming conventions require discipline on the part of a modeler or a team of modelers. Comments are also helpful for documenting secondary data.

3.2 LINK AND ASSOCIATION CONCEPTS

3.2.1 Multiplicity

In Chapter 2, we introduced the notion of multiplicity. For database applications it is helpful to think in terms of minimum multiplicity and maximum multiplicity.

Minimum multiplicity is the lower limit on the possible number of related objects. Figure 3.9 shows several examples; the most common values are zero and one. We can implement a minimum multiplicity of zero by permitting null values and a minimum multiplicity of one by forbidding null values. A minimum multiplicity greater than one often requires special programming; fortunately such a minimum multiplicity seldom occurs.

OMT construct	*Minimum multiplicity*	*Maximum multiplicity*
Class	*1*	*1*
Class	*0*	*infinite*
Class	*0*	*1*
1..* Class	*1*	*infinite*
2..4 Class	*2*	*4*

Figure 3.9 Examples of minimum and maximum multiplicity

A minimum multiplicity of one or more implies an existence dependency between objects. In our airline flight example (Figure 2.23) a flight reservation concerns one flight and a flight may be reserved by many flight reservations. *Flight* has a minimum multiplicity of one in this association. It does not make much sense to make a flight reservation unless you refer to a flight. Furthermore, if the airline cancels a flight, it must notify all passengers with a corresponding flight reservation. In contrast, *FlightReservation* has a minimum multiplicity of zero in this association. You can add a flight without regard for flight reservations. Similarly, the airline can cancel a flight reservation without affecting a flight.

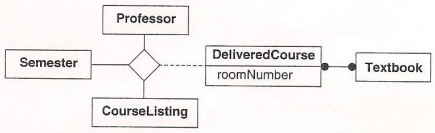
Maximum multiplicity is the upper limit on the possible number of related objects. The most common values are one and infinite.

The choice of multiplicity depends on the application. For a multiplicity of exactly one, consider whether the target object is truly mandatory or possibly optional; use a hollow ball to indicate an optional target object. Furthermore, the target could be a set of objects rather than a single object; use a solid ball to allow a set of objects. The proper choice of multiplicity depends on application requirements. Be sure to review carefully whatever multiplicity decisions you make.

3.2.2 Ternary Associations

The ***degree of an association*** is the number of roles for each link. Associations may be binary, ternary, or higher degree. The vast majority are binary or qualified binary, and we described them in Chapter 2. Ternary associations occasionally occur, but we have rarely encountered an association of higher degree.

A ***ternary association*** is an association with three roles that cannot be restated as binary associations. The notation for a ternary association is a large diamond; each associated class connects to a vertex of the diamond with a line. In Figure 3.10 a professor teaches a listed course for a semester. The delivered course may use many textbooks; the same textbook may be used for multiple delivered courses. A ternary association may have link attributes or be treated as an association class, as in Figure 3.10.



{Candidate key for ternary association = (semesterID, professorID, courseListingID)}

Figure 3.10 Ternary associations occasionally occur in models

3.2.3 Candidate Keys for Associations

Note that there are no balls next to the diamond or the classes of the ternary association in Figure 3.10. Although we could extend multiplicity notation to accommodate ternary associations, we prefer to use candidate keys to avoid confusion. A ***candidate key*** for an association is a combination of roles and qualifiers that uniquely identifies links within an association. Since the roles and qualifiers are implemented with attributes, we use the term "candidate key" for both classes and associations. The collection of roles and qualifiers in a candidate key must be minimal; no role or qualifier can be discarded from the candidate key without destroying uniqueness. Normally a ternary association has a single candidate key

that is composed of roles from all three related classes. Occasionally you will encounter a ternary association with a candidate key that involves only two of the related classes.

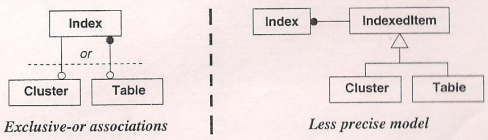
The combination of *semesterID*, *professorID*, and *courseListingID* is a candidate key for *DeliveredCourse*. A professor may teach many courses in a semester and many semesters of the same course; a course may be taught by multiple professors. The confluence of semester, professor, and course listing is required to identify uniquely a delivered course.

We indicate a candidate key for an association with a comment in braces.

3.2.4 Exclusive-Or Associations

An ***exclusive-or association*** is a member of a group of associations that emanate from a class, called the ***source*** class. For each object in the source class exactly one exclusive-or association applies. An exclusive-or association relates the source class to a ***target*** class. An individual exclusive-or association is optional with regard to the target class, but the exclusive-or semantics requires that one target object be chosen for each source object. An exclusive-or association may belong to only one group.

Figure 3.11 shows an example in which *Index* is the source class and *Cluster* and *Table* are target classes. This example is an excerpt from the model for the *Oracle* relational DBMS. An index is associated with a table or a cluster, but not both, so a dashed line annotated by *or* cuts across the association lines close to the target classes. Interpreting the ball notation, a table may have zero or more indexes while a cluster has exactly one index. The alternative model using generalization is less precise and loses a multiplicity constraint: In the left model a cluster is associated with one index; in the right model a cluster can be associated with many indexes (via inheritance).



Exclusive-or associations *Less precise model*

Figure 3.11 Exclusive-or associations can yield a more precise model

3.2.5 Qualified Associations

In Section 2.2.6, we introduced the notion of qualification and presented the most common situation, a single qualifier that reduces the maximum multiplicity of the target role from "many" to "one." Qualification does not affect the minimum multiplicity of an association. We now present more complex forms of qualification.

A qualifier selects among the objects in the target set and usually, but not always, reduces effective multiplicity from "many" to "one." In Figure 3.12 "many" multiplicity still remains after qualification. A company has many corporate officers, one president, and one treasurer but many directors and many vice presidents. Therefore, the combination of a company and office can yield many persons.

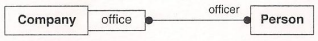


Figure 3.12 Qualification need not yield a target multiplicity of "one"

A ***qualification cascade*** is a series of consecutive qualified associations. Qualification cascades are encountered where an accumulation of qualifiers denotes increasingly specific objects. For example, in Figure 3.13 a city is identified by the combination of a country name, state name, and city name.

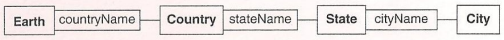


Figure 3.13 A qualification cascade denotes increasingly specific objects

A ***compound qualifier*** consists of two or more attributes that combine to refine the multiplicity of an association. The attributes that compose the compound qualifier are "anded" together. Figure 3.14 shows a compound qualifier. Both the node name and edge name are required to locate a connection for a directed graph.

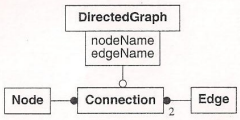


Figure 3.14 A compound qualifier refines the multiplicity of an association

3.3 AGGREGATION

Aggregation is a kind of association, between a whole, called the ***assembly***, and its parts, called the ***components*** [Blaha-93b]. Aggregation is often called the "a-part-of" or "parts-explosion" relationship and may be nested to an arbitrary number of levels. Aggregation bears the transitivity property: If *A* is part of *B* and *B* is part of *C*, then *A* is part of *C*. Aggregation is also antisymmetric: If *A* is part of *B*, then *B* is not part of *A*. Transitivity lets you compute

the transitive closure of an assembly—that is, you can compute the components that directly and indirectly compose it. ***Transitive closure*** is a term from graph theory; the transitive closure of a node is the set of nodes that are reachable by some sequence of edges.

As Figure 3.15 shows, aggregation is drawn like an association with a small diamond added next to the assembly end. A book consists of front matter, multiple chapters, and back matter. Front matter, in turn, consists of a title page and a preface; back matter consists of multiple appendixes and an index.

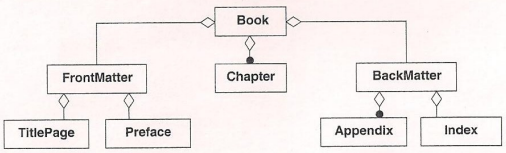


Figure 3.15 Aggregation is a kind of association with additional semantic properties

Having presented this example, we hasten to add that aggregation does not specify order. An assembly is a collection of components without any implied order. It just so happens that a book has a well-known and simple order of parts; the back matter follows the chapters, which follow the front mater. If the model had to capture component order, you would have to include comments, as we describe later in Section 3.7, or use a generic class, as described in Chapter 4.

Aggregation is often encountered with problems that involve bills-of-material. A ***bill-of-material*** is a report that lists each part on a separate line; the lines of the report are ordered by traversing the components in depth-first order starting from the root assembly. Each line may be indented according to its level in the hierarchy. Sibling parts (parts with the same parent) may be further ordered by some other criteria. Figure 3.16 shows a bill-of-material with two levels of parts. In practice, bills-of-material are often nested more deeply.

An aggregation relationship is essentially a binary association, a pairing between the assembly class and a component class. An assembly with many kinds of components corresponds to many aggregations. We define each individual pairing as an aggregation so that we can specify the multiplicity of each component within the assembly. This definition emphasizes that aggregation is a special form of association. An aggregation can be qualified, have roles, and have link attributes just like any other association.

For bill-of-material problems the distinction between association and aggregation is clear. However, for other applications it is not always obvious if an association should be modeled as an aggregation. To determine if an association is an aggregation, test whether the "is-part-of" property applies. The asymmetry and transitivity properties must also hold for aggregation. When in doubt about whether association or aggregation applies, the distinction is not important and you should just use ordinary association.

Bill-of-Material

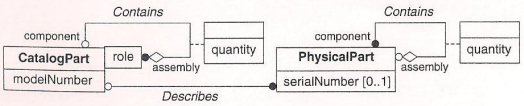
Level	Part num	Name	Quantity
01	LM16G	Lawn mower	1
02	B16M	Blade	1
02	E1	Engine	1
02	W3	Wheel	4
02	D16	Deck	1

Figure 3.16 Aggregation often occurs with problems that involve bills-of-material

3.3.1 Physical versus Catalog Aggregation

It is important to distinguish between physical and catalog aggregation. ***Physical aggregation*** is an aggregation for which each component is dedicated to at most one assembly. ***Catalog aggregation*** is an aggregation for which components are reusable across multiple assemblies. As an example, consider physical cars (items with individual serial numbers) and car models (Ford Escort, Mazda 626). Customer service records refer to physical cars, while design documents describe car models. The parts explosion for a physical car involves physical aggregation, and the parts explosion for a car model involves catalog aggregation.

Figure 3.17 shows the canonical relationship between catalog aggregation and physical aggregation. A catalog part may describe multiple physical parts. Each catalog part and physical part may contain lesser parts. A catalog part may belong to multiple assemblies, but a physical part may belong to at most one assembly. (The text in braces is a constraint, which we describe later in Section 3.7.)



{The catalog parts and physical parts aggregations must be acyclic.}

Figure 3.17 Relationship between catalog aggregation and physical aggregation

Catalog parts may have a quantity specified within a context of usage. For example, two screws of a given type may be used for the windshield wiper assembly of a car and four screws of the same type used for the glove box assembly. A role, such as windshield wiper assembly and glove box assembly, may be specified to differentiate the various uses of a part. A series of roles provides a unique path for navigating a catalog-aggregation lattice.

A physical aggregation tree may also have quantities specified for parts. Parts with individual serial numbers always have a quantity of one, since each part must be individually noted. In contrast, other parts are interchangeable, such as nuts and bolts taken from a bin.

Interchangeable physical parts have identity in the real world, but the corresponding physical aggregation model may not preserve this identity.

The instances for physical aggregation form a collection of trees. Each part belongs to at most one assembly. The part at the root of the tree does not belong to any assembly, and all other parts within the tree belong to exactly one assembly.

In contrast, the instances for catalog aggregation form a directed acyclic graph. (The term "acyclic" means that you cannot start with a part and traverse some sequence of components and reach the starting part.) An assembly may have multiple components and a component may belong to multiple assemblies, but there is a strict sense of direction concerning which part is the assembly and which part is the component (antisymmetry).

The notation clearly indicates whether physical or catalog aggregation applies. With physical aggregation the assembly class has a multiplicity of "one" or "zero or one." With catalog aggregation the assembly class has a multiplicity of "many."

3.3.2 Extended Semantics for Physical Aggregation

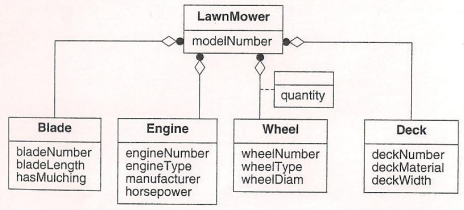
Physical aggregation bears properties in addition to transitivity and antisymmetry.

- **Propagation of operations**. ***Propagation*** is the automatic application of some property to a network of objects when the property is applied to some starting object. With aggregation some operations of the assembly may propagate to the components with possible local modifications. For example, moving a window moves the title, pane, and border. For each operation and other propagated qualities, you may wish to specify the extent of propagation [Rumbaugh-88].
- **Propagation of default values**. Default values can also propagate. For example, the color of a car may propagate to the doors. It may be possible to override default values for specific instances. For example, the color of a door for a repaired car may not match the body.
- **Versioning**. A ***version*** is an alternative object relative to some base object. You can encounter versions with hypothetical situations, such as different possibilities for an engineering design. With aggregation, when a new version of a component is created, you may want to trigger automatically the creation of a new version of the assembly.
- **Composite identifiers**. The identifier of a component may or may not include the identifier of the assembly.
- **Physical clustering**. Aggregation provides a basis for physically clustering objects in contiguous areas of secondary storage for faster storage and retrieval. Components are often accessed in conjunction with an assembly. Composite identifiers make it easier to implement physical clustering.
- **Locking**. Many database managers use locking to facilitate concurrent, multiuser access to data. The database manager automatically acquires locks and resolves conflicts without any special user actions. Some database managers implement efficient locking for aggregate trees: A lock on the assembly implies a lock on all components. This is more efficient than placing a lock on each affected part.

3.3.3 Extended Semantics for Catalog Aggregation

Catalog aggregation has fewer properties than physical aggregation, but it is still important to recognize so that you do not confuse it with physical aggregation. For example, propagation is not helpful with catalog aggregation, because a component could have multiple, conflicting sources of information. Propagation across catalog aggregation is too specialized and unusual a topic for us to devise a general solution. Catalog aggregation still observes the basic properties of transitivity and antisymmetry.

With catalog aggregation a collection of components may imply an assembly. This situation is commonly encountered with structured part names in bills-of-material. Figure 3.18 shows a hypothetical object diagram for a lawn mower with two sample bills-of-material. The model number of a lawn mower is a structured name consisting of the prefix "LM" followed by two characters indicating the blade length followed by one character denoting a gas or electric engine. In this example, the blade length and engine type are sufficient to identify a lawn mower uniquely.



Bill-of-Material

Level	Part num	Name	Quantity
01	LM16G	Lawn mower	1
02	B16M	Blade	1
02	E1	Engine	1
02	W3	Wheel	4
02	D16	Deck	1

Bill-of-Material

Level	Part num	Name	Quantity
01	LM16E	Lawn mower	1
02	B16M	Blade	1
02	E1E	Engine	1
02	W3	Wheel	4
02	D16	Deck	1

Figure 3.18 Structured part names for catalog aggregation

Note that the object model does not capture the constraint that a collection of components implies one assembly. The object diagram in Figure 3.18 states that a lawn mower has "many" multiplicity with respect to each component. In other words, the engine design is useful for multiple lawn mower designs; a blade design applies to multiple lawn mower designs; and so on. You could add a comment to the object model if you wanted to note that some components taken together imply a single lawn mower design.

3.4 GENERALIZATION

3.4.1 Abstract and Concrete Classes

A ***concrete class*** is a class that can have direct instances. In Figure 3.19 *Stock*, *Bond*, and *Insurance* are concrete classes because they have direct instances. *FinancialInstrument* also is a concrete class because some *FinancialInstrument* occurrences (such as real estate) are not in the listed subclasses. The legend *concrete* below the class name indicates a concrete superclass.

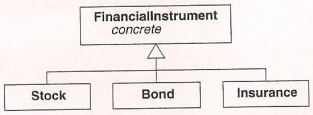


Figure 3.19 Concrete classes

An ***abstract class*** is a class that has no direct instances. The descendant classes can also be abstract, but the generalization hierarchy must ultimately terminate in subclasses with direct instances. In Figure 3.20 *Person* is an abstract class but the subclasses *Manager* and *IndividualContributor* are concrete. The legend *abstract* indicates an abstract superclass. You may define abstract operations for abstract classes. An ***abstract operation*** specifies the signature of an operation while deferring implementation to the subclasses. The ***signature*** of an operation specifies the argument types, the result type, exception conditions, and the semantics of the operation. The notation for an abstract operation is the legend *{abstract}* following the operation name.

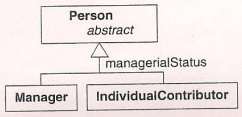


Figure 3.20 An abstract class

3.4.2 Generalization versus Other Object-Modeling Constructs

Figure 3.21 shows how generalization differs from association. Both generalization and association involve classes, but association describes the relationship between two or more instances, while generalization describes different aspects of a single instance.

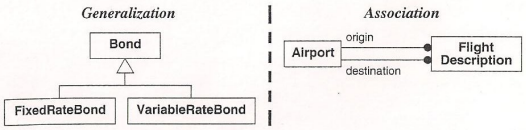


Figure 3.21 Generalization versus association

During analysis, associations are more important than generalization. Associations add information. Associations transcend class encapsulation boundaries and can have a broad impact on a system. In contrast, generalization eliminates duplications of shared properties (consolidates), but does not change the instances that conform to the model.

During design, generalization becomes more significant. Developers tend to discover data structure during analysis and behavior during design. Generalization provides a reuse mechanism for concisely expressing behavior and including code from class libraries. Judicious reuse reduces development time and substitutes carefully tested library code for error-prone application code.

Figure 3.22 shows how generalization differs from aggregation. Both generalization and aggregation give rise to trees through transitive closure, but generalization is the "or" relationship and aggregation is the "and" relationship. In the figure, a bond is a fixed-rate bond *or* a variable-rate bond. A book comprises front matter *and* many chapters *and* back matter. Generalization relates classes that describe different aspects of a single object. Aggregation relates distinct objects that compose an assembly.

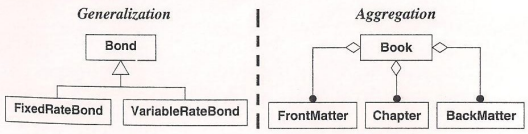


Figure 3.22 Generalization versus aggregation

Figure 3.23 shows how generalization differs from instantiation. Some modelers erroneously introduce an object as a subclass in a generalization. Generalization does not deal with individual objects; generalization relates *classes*—the superclass and the subclasses. Instantiation relates an instance to a class.

A common mistake is to confuse a subclass with a role. Figure 3.24 shows the difference between a subclass and a role. A subclass is a specialization of a class; a role is a usage of a class. By definition, a subclass pertains to only some superclass instances. There is no such constraint with a role; a role may refer to any or all instances. Figure 3.24 shows roles and

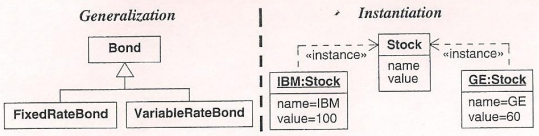


Figure 3.23 Generalization versus instantiation

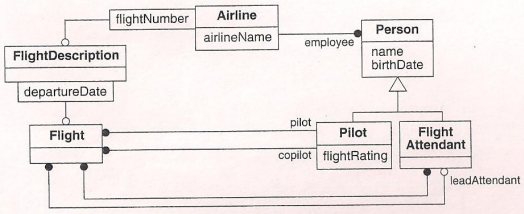


Figure 3.24 Subclass versus role

subclasses for the class *Person*. *Pilot* and *FlightAttendant* are modeled as subclasses, since they have different attributes and associations. In contrast, *employee* is merely a role for a *Person*; *pilot* and *copilot* are roles for *Pilot*; and *leadAttendant* is a role for *FlightAttendant*.

3.5 MULTIPLE INHERITANCE

Multiple inheritance permits a class to inherit attributes, operations, and associations from multiple superclasses, which, in turn, lets you mix information from two or more sources. Single inheritance organizes classes as a tree. Multiple inheritance organizes classes as a directed acyclic graph. Multiple inheritance brings greater modeling power but at the cost of greater complexity.

3.5.1 Multiple Inheritance from Different Discriminators

Multiple inheritance can arise through different bases (different discriminators) for specializing the same class. In Figure 3.25 a person may be specialized along the bases of manage-

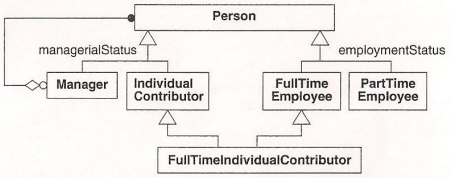


Figure 3.25 Multiple inheritance from different discriminators

rial status (manager or individual contributor) and employment status (fulltime or parttime). Whether a person is a manager or not is independent of employment status. Four subclasses are possible that combine managerial status and employment status. The figure shows one, *FullTimeIndividualContributor*.

3.5.2 Multiple Inheritance without a Common Ancestor

Multiple inheritance is possible, even when superclasses have no common ancestor. This often occurs when you mixin functionality from software libraries. When software libraries overlap or contradict, multiple inheritance becomes problematic.

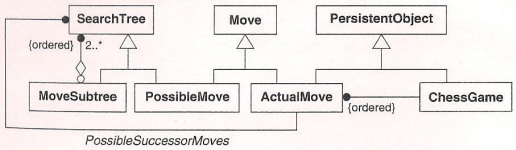


Figure 3.26 Multiple inheritance without a common ancestor

Figure 3.26 shows an excerpt of a model for a chess game. A chess program looks ahead from the current board position and examines multiple search trees to determine the most promising next move for actual play. You may wish to store the actual moves in a database for reconsideration at some later date or in order to replay a game from an intermediate point. In contrast, the exploration of the search space may be regarded as transient and unimportant to store.

In Figure 3.26 each *SearchTree* may be a *MoveSubtree* or a *PossibleMove*. Each *MoveSubtree*, in turn, is composed of lesser *SearchTrees*. Such a combination of object-modeling

constructs can describe a tree of moves of arbitrary depth. Each *Move* may be a *PossibleMove* or an *ActualMove*. *PossibleMove* and *ActualMove* inherit common behavior from the *Move* superclass. *ActualMove* and *ChessGame* are the classes with the objects that we want to store permanently. In our model persistent objects must inherit from *PersistentObject*, as with the ODMG standard [Cattell-96]. In Figure 3.26 both *PossibleMove* and *ActualMove* use multiple inheritance.

3.5.3 Workarounds for Multiple Inheritance

Several workarounds are possible if you wish to avoid the complexity of multiple inheritance. Workarounds often make it easier to understand and implement a model.

- **Factoring**. Figure 3.27 avoids multiple inheritance by taking the cross product of the orthogonal bases of person (*managerialStatus*, *employmentStatus*) from Figure 3.25. The disadvantages are that you lose conceptual clarity and the reuse of similar code is more cumbersome.

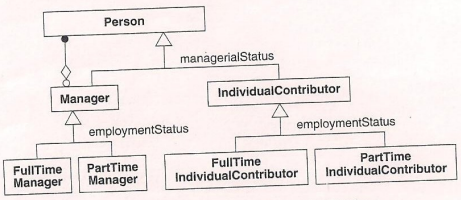


Figure 3.27 Multiple inheritance workaround: factoring

- **Fragmenting a subclass**. You may fragment a subclass into multiple classes—one for each superclass. For example, we could restate Figure 3.26 as Figure 3.28. The disadvantages are that objects are broken into multiple pieces and the added classes are often artificial and difficult to define.
- **Replacing generalization with associations**. You can replace a generalization with several exclusive-or associations. Figure 3.29 shows the result of doing this to the model in Figure 3.26. This option is most viable for a generalization with few subclasses. The disadvantages are the loss of identity and more cumbersome reuse of operations. Also, exclusive-or associations are not supported by most languages and database managers, so you may need to write application code to enforce exclusivity.

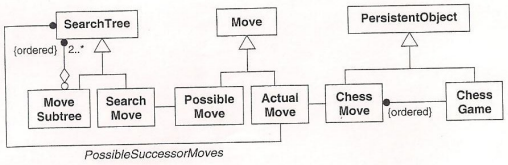


Figure 3.28 Multiple inheritance workaround: fragmenting a subclass

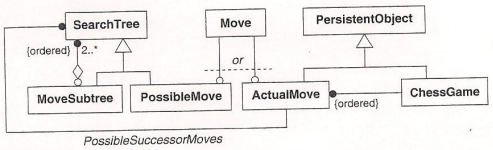


Figure 3.29 Multiple inheritance workaround: replacing generalization with exclusive-or associations

3.6 PACKAGES

You can fit an object model on a single page for many small and medium-sized problems. However, you will need to organize the presentation of large object models. A person cannot understand a large object model at a glance. Furthermore, it is difficult to get a sense of perspective about the relative importance of portions of a large model. You must partition a large model to allow comprehension.

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 3.30 shows, the notation for a package is a box with the addition of a tab. The purpose of the tab is to suggest the enclosed contents, like a tabbed folder.

There are various themes for forming packages: dominant classes, dominant relationships, major aspects of functionality, and symmetry. For example, many business systems

across for the third hand. No cards are passed for the fourth hand. The fifth hand starts the passing cycle over again with the left player. Each player chooses the cards for passing. A good player will assess a hand and try to pass cards that will reduce his or her own likelihood of scoring points and increase that for the receiving player.

The card-playing portion of a hand consists of 13 tricks. The player with the two of clubs leads the first trick and play continues clockwise. The player who plays the largest card of the lead suit "wins" the trick and adds the cards in the trick to his or her pile for scoring at the end of the hand. The winner of the trick also leads the next trick. The sequence of cards in a suit from largest to smallest is ace, king, queen, jack, and ten down to two. Each card is played exactly once in a game.

If possible, a player must play the same suit as the lead card on a trick. Furthermore, on the first trick, a player cannot play a card that scores points (see next paragraph), unless that is the only possible play. For subsequent tricks, a player without the lead suit can play any card. A player cannot lead hearts until they have been broken (a heart has been thrown off-suit on a preceding trick) or only hearts remain in the player's hand.

At the end of a hand, each player's pile is scored. The queen of spades is 13 points; each heart counts one point; all other cards are zero points. Ordinarily, a player's game score is incremented by the number of points in that player's pile. The exception is a shoot, when one player takes the queen of spades and all the hearts. The player accomplishing a shoot receives zero points, and all opponents receive 26 points. The game ends when one or more players have a game total of at least 100 points.

4

Object Metamodeling

In this chapter we discuss object metamodeling, which you will need for some very advanced applications. You should be proficient with the material in Chapters 2 and 3 before continuing with this chapter. Those doing an initial reading can skip this chapter.

4.1 METADATA AND METAMODELS

Data is information about application concepts and relationships; a ***model*** is an abstraction of data. Similarly, ***metadata*** is information about a model and a ***metamodel*** is an abstraction of metadata. The relationship of metadata to data is relative, and you can continue the abstraction process to create models at successively higher levels, as Figure 4.1 shows. The lowest level of abstraction is data—information that directly relates to an application. Data is abstracted into classes, relationships, events, states, and operations that can, in turn, be abstracted into metaconstructs.

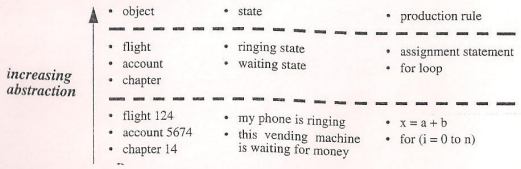


Figure 4.1 Levels of abstraction

Metamodels arise for some complex applications, such as DBMSs and model editors. DBMSs facilitate a wide variety of applications that manage persistent data. The data dictionary of a DBMS can span many applications and thus must deal with metadata. A model editor, such as an object model editor, can also handle many applications. Since a model editor is an application in its own right, you can describe it with a model, in this case a metamodel. Parts catalogs, blueprints, and dictionaries are other examples of metadata.

Figure 4.2 shows a progression of data and models in order of increasing abstraction. The employee data and model could arise for an application like payroll processing. The metadata and metamodel are an excerpt from an object model editor that could be used to construct the application model. Of course, this is not the only possible metamodel. The contents of a metamodel, as for any other application, depend on the problem requirements and the skill and style of the modeler.

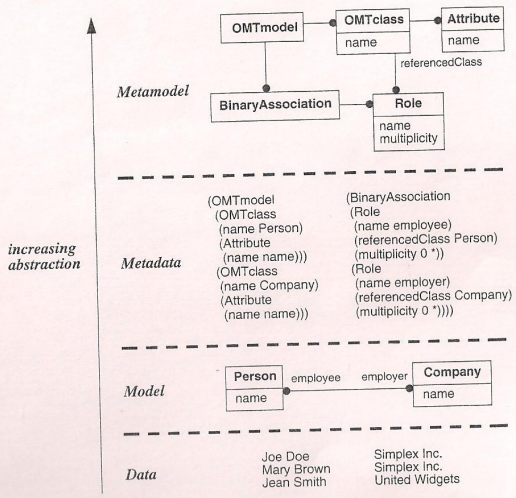


Figure 4.2 An example of data and model, metadata and metamodel

There are many reasons for using metamodels. Metamodels promote reuse by making software serve a category of applications. An application can be conceived broadly to satisfy multiple needs. Expert system shells, DBMSs, software development tools, and the UNIX tools LEX and YACC are examples of software that intrinsically involve metamodeling. Metamodels also isolate software from changes in an application model, in effect detaching the software from the details of an application.

However, metamodels have some disadvantages. They are complex, making them confusing to understand and difficult to debug. A developer must experience several simpler applications before taking the intellectual leap of abstraction that metamodels require. Also, metamodel-driven software may perform more slowly than custom software. For example, a DBMS provides general data management routines. You could perform faster data management by custom coding the data access routines for an application. However, the tradeoff for the performance overhead of a DBMS is the savings in development time and effort.

Languages and DBMSs 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 Ada, C, C++, and Eiffel deal with metadata at compile-time but do not make the metadata explicitly available at run-time. Relational DBMSs incorporate a data dictionary that describes the structure of all tables, attributes, indexes, and other system objects, including the data dictionary tables. Programmers can read and write to the data dictionary at run-time. Object-oriented DBMSs also make metadata available at run-time, but access can be complicated by programming language restrictions.

4.1.1 Generic Classes

A ***generic class*** is a class that combines data and metadata. In Figure 4.3 and Figure 4.4 we have replaced application classes with generic classes. In Figure 4.3 *WindowParameter* is a generic class that pairs the name of a window default attribute with its corresponding value. The generic *DocumentComponent* class in Figure 4.4 eliminates the *Page*, *Paragraph*, and *Line* classes, capturing the distinction with the *documentType* attribute.

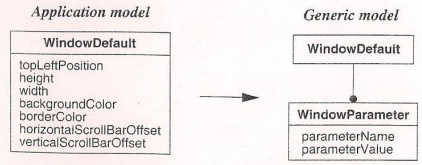


Figure 4.3 Making attributes generic: Application model versus generic model

Generic classes have several advantages. They can often simplify a model; a few generic classes and attributes can substitute for many application classes and attributes. Generic

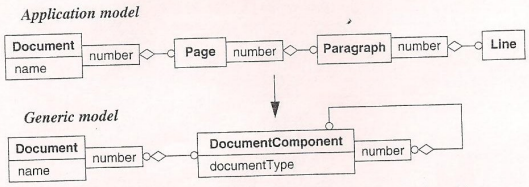


Figure 4.4 Making classes generic: Application model versus generic model

classes let you accommodate classes and attributes that may not be known as the application is written. Application software need only conform to the loose generic model rather than to a more rigid application model. For example, adding a window default parameter to a generic model would not change the application code, but if you changed an application model, you would have to modify and recompile the application code. Generic classes let you define objects at run-time, even in environments with compile-time type checking.

Generic classes also have their disadvantages. The metadata complicates programming. With an application model, you represent the various application attributes directly as programming variables. With a generic model, you must generate application code with a preprocessor at compile-time or query a class at run-time to determine the attributes and their values. Method code is more difficult to write for generic data structures than for application data structures. Finally, generic classes can slow performance because of the extra processing required to bind a variable to a value.

In summary, generic classes provide a valuable modeling technique that is occasionally helpful. The increased complexity and performance overhead can be offset by greater brevity and extensibility.

4.1.2 Reification

Reification is the promotion of something that is not an object into an object [Rumbaugh-92]. 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.

In Figure 4.5 we have promoted the *substanceName* attribute to a class to capture the "many-to-many" relationship between *Substance* and *SubstanceName*. Each chemical substance may be referenced by 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*. Exercise 4.8 also illustrates reification.

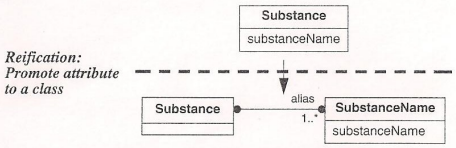


Figure 4.5 Reification: Promotion of an attribute to a class

4.1.3 OMT Object Metamodel

Figure 4.6 presents an abridged OMT object metamodel. Object models embody three major constructs: class, association, and generalization. Our metamodel recognizes the similarities between classes and associations with the *Element* superclass. Both classes and associations have names, deal with instances, may participate in associations, and have candidate keys. The difference is that objects in a class have intrinsic identity, while the identity of links in an association is derived from the related elements. You should try to maintain the uniqueness of element names within an object model.

An element is described by zero or more instance attributes. Each attribute has a name. An attribute also has attribute multiplicity and a domain once a model is fully specified during design. As noted in the model, the name of an instance attribute is unique within the scope of a class or association but need not be globally unique. Enumeration domains have multiple enumeration values. A domain may be structured and consist of lesser domains.

An element may assume different roles in relationships with other elements. Role names may not collide with attribute names for a given element. (See Section 2.2.3.) The metamodel includes qualified associations because a role may involve qualifiers. Each element may have multiple candidate keys. Each candidate key consists of one or more roles, qualifiers, and instance attributes.

Generalization and association are the primary relationships in OMT object modeling. Both generalization and association involve two or more classes. The difference is that association describes the relationship between two or more instances; generalization describes different aspects of a single instance. Aggregation is a special kind of association and consequently also describes a relationship between instances.

Figure 4.6 supports multiple inheritance because a class can participate in multiple generalizations as a superclass and multiple generalizations as a subclass. The *abstractOrConcrete* attribute indicates if every object must be further described by a subclass. (See Section 3.4.1.) Each generalization may have a discriminator. The discriminator is an enumeration attribute that indicates which subclass further describes a given superclass object.

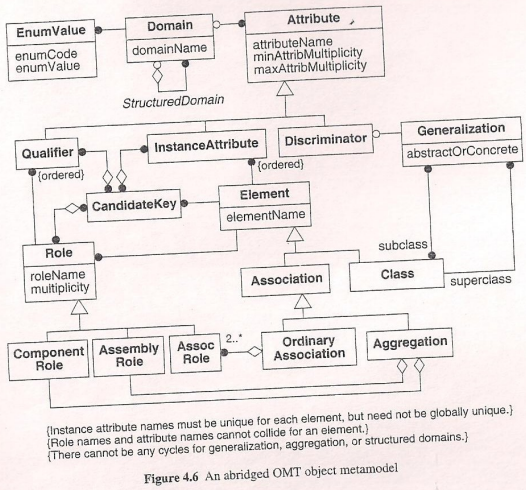


Figure 4.6 An abridged OMT object metamodel

4.2 FRAMEWORKS

A *framework* [Johnson-88] is a skeletal structure of a program that must be elaborated to build a complete application. This elaboration often consists of specializing abstract classes with behavior specific to an individual application. A framework may be accompanied by a class library so that the user can perform much of the specialization by choosing the appropriate subclasses rather than programming subclass behavior from scratch. Frameworks consist of more than just the classes involved and include a paradigm for flow of control and shared invariants. Frameworks tend to be specific to a category of applications; framework class libraries are often application specific and not suitable for general use.

[Blaha-90] describes a framework that generates bills-of-material from functional requirements. The bill-of-material generator has no class libraries. Instead it consists of generic code that is customized with an object model and decision table definitions for the desired application. The framework includes a declarative language to compensate for the limitations of decision tables. Flow of control is implicit in the decision tables and declarative programming code. The bill-of-material generator is intended for applications that configure a mass-produced mechanical assembly from a modest number of constituent parts that mix and match in many ways. Thus the bill-of-material generator would be suitable for lawn mowers (mass production) but would not apply to the space shuttle (few produced, too many kinds of component parts).

The EXODUS extensible database manager [Carey-86] is another example of a framework. The goal of EXODUS is to "facilitate the fast development of high-performance, application-specific database systems." An increasing variety of applications require database services and the EXODUS developers believe a single DBMS will be unable to satisfy the diverse requirements. The EXODUS framework includes generic code, class libraries, and a programming language specially designed for writing database system software. Application database managers with either a relational flavor or an object-oriented flavor can be developed with EXODUS.

Frameworks are important not only for their ability to organize large systems, but also for their contribution to reuse. A framework supports reuse at a larger granularity than classes, promoting reuse of designs instead of code. Frameworks can be built on other frameworks, yielding a hierarchy of abstractions.

However, frameworks are difficult to build and understand. They often involve metamodels and generic classes, both of which are intrinsically complex. You should experience several applications before attempting a framework. Tangible applications are easier to grasp than the abstractions of a framework. With a framework it is difficult to foresee all requirements, exceptions, and deviations; the inevitable peculiarities of applications stress a framework.

4.3 PATTERNS

A *pattern* is an excerpt of an object model with one or more parameters as placeholders for classes and associations. You incorporate a pattern into a model by substituting specific classes and associations for the parameters, that is by instantiating the pattern.

Most of the literature, including recent OOPSLA conference workshops and an Internet pattern user group, has been concerned with *design* patterns [Gamma-95] and *implementation* patterns, called "idioms" in [Coplien-92]. In this section we focus on analysis patterns. Analysis patterns are more relevant to databases, while design and implementation patterns are more relevant to programming. Some frequently encountered analysis patterns are tree, graph, item description, and homomorphism. We describe each of these in detail later.

There are many benefits to using patterns in model construction:

- **Enriched OMT language**. Patterns provide a higher level of building blocks for models than the base primitives of class, association, and generalization. Patterns are prototypical modeling fragments that distill some of the knowledge of experts.

The system should let us browse the database directly (such as with interactive DBMS commands), but discourage direct database update. All updates should occur via the user interface.

We will implement portions of the portfolio manager with files, a relational DBMS, and an object-oriented DBMS for illustrative purposes. It would be acceptable just to store the data in files. However, we would prefer to store all the data in a formal database to facilitate ad hoc queries and extensibility.

The next chapter shows the resulting problem statement for the portfolio manager.

7.3 CHAPTER SUMMARY

Conceptualization is the first phase of the OMT process for software development. Conceptualization begins with a person thinking of an idea for a new system to provide new functionality or simplify delivery of existing functionality. The output of conceptualization is a statement of specific requirements that serve as grist for the subsequent phase of analysis.

BIBLIOGRAPHIC NOTE

James Rumbaugh first sensitized us to the need for a conceptualization stage that precedes analysis.

8

Analysis

The purpose of analysis is to understand the problem requirements thoroughly and devise a model of the real world. During analysis the developer describes *what* must be done, not *how* it should be done. You must understand a problem before attempting a solution. The models constructed during analysis provide the basis for subsequent design and implementation. The OMT analysis process applies to various implementation targets, including databases and programming code.

Generally, you need all three OMT models—object, dynamic, and functional. However, a particular application may be unbalanced and not fully exercise all three models. In fact, most database applications lack a significant dynamic model. The case study that we introduced in the preceding chapter (portfolio manager) has a rich and robust object model, a weak dynamic model, and a modest functional model.

8.1 OVERVIEW

Various sources of information can serve as inputs to analysis. Often you will have a formal statement that describes your problem. In this book we use a concise problem statement; in commercial settings you may encounter RFPs (requests for proposal) and RFQs (requests for quotation). Sometimes domain experts will provide scenarios, storyboards, and use cases for a new system. (A ***use case*** [Jacobson-92] is a theme for interacting with a system. See Section 8.6.) You may also have access to problem experts for interviews and advice. On fortunate occasions, you yourself may be a source of problem expertise. Replacements for existing systems can benefit from prior artifacts: manuals, data entry forms, and other sources of grist for reverse engineering (see Chapter 20). We aggressively seek all existing information at the start of a project.

The outputs from analysis are three kinds of models that capture crucial aspects of a system. The object model describes objects and their relationships. The dynamic model characterizes temporal interactions of objects and their responses to events. The functional model

defines operations and constraints. The data dictionary serves as a detailed repository for the three models, documenting names, concepts, definitions, examples, and rationale. You should also prepare a textual narrative to guide the reader through the models.

We have included use cases as part of the OMT methodology. (See Section 8.6.) But we have given use cases less prominence than some readers might expect. Use cases are important because they specify requirements for interaction in a manner that is understandable to both the software developer and the business client. But use cases are only one source of information for system development. We desire to exploit *all* information sources—including statements of requirements, existing documentation, and reverse engineering. Furthermore, our experience has been that artifacts for reverse engineering are normally available for database applications, as well as statements of requirements. We encounter use cases less often.

8.2 PROBLEM STATEMENT

Always begin analysis with a written problem statement. The statement need not be rigorous or complete, but you should at least have the motivation for a new system in writing. The problem statement should focus on the minimal functionality needed to make a useful system. You can always add functionality later. Carefully delimit the scope of a new system from other applications with which it will interact. Keep in mind that the problem statement is only a starting point and you will not fully know the true requirements until after analysis.

Beware of problem statement pitfalls. Problem statements are often ambiguous. Natural language is expressive, but it is also vague and subject to multiple interpretations. Problem statements often contain errors. When working with natural language, most persons have difficulty grasping the full scope of a problem and recognizing inconsistencies. Problem statements may be underspecified and lack important information. Similarly, a problem statement may be overspecified and contain details that would be better left to design. You must be careful not to accept every thought and every word literally; instead apply reasoned judgment and act upon perceived intent.

Figure 8.1 is the problem statement for the portfolio manager case study (described in Chapter 7). The examples (they are real) help clarify the requirements, but do not convey additional requirements.

8.3 OBJECT MODEL

As we stated earlier, the object model is the most stable part of a system. The object model defines the universe of discourse for the other two models and provides the lattice to which behavior is attached. Objects are the things that interact and perform computations.

Within the object model the class-association structure is critical, and you should develop this first. Only then should you add inheritance—so that you do not prematurely bias the model. You will discover operations as a byproduct of dynamic and functional modeling.

Object modeling begins with an analysis of the problem statement and has the following steps. (The numbers in brackets refer to sections that explain these steps in detail.)

Develop software for managing investment portfolios. The following capabilities must be provided:

- Permit a mixture of assets and liabilities, including bonds, stocks, options, commodities, mutual funds, precious metals, collectibles, cash, insurance, real estate, and loans. A financial instrument is an asset, liability, or portfolio.
- Support hierarchies of portfolios. Portfolios contain financial instruments.
- Record dates of purchase and sale of financial instruments and their price. Allow for stock splits and issuance of different kinds of securities.

  Example: In 1991 KV Pharmaceuticals issued one share of class A stock for each existing share. Existing shares were designated class B stock. Class A stock has a higher dividend; class B stock is supervoting. (Supervoting stock has additional votes per share.)

  Example: In 1993 the Chile Fund issued one warrant for each share of stock. Seven warrants entitled the holder to purchase two shares of stock at $26 per share. The warrants expired about one month after their issuance. The warrants traded on the New York Stock Exchange as well as the primary stock.
- Track value of assets over time.
- Handle dividends, both stock and cash. Handle interest payments.

  Example: In 1992 the Mexico Fund issued both cash dividends and stock dividends.
- Compute ROI.
- Handle currency conversions with possible fees.
- The developer should look at broker statements, stock exchange listings, existing financial programs, and other sources of information in order to discover relevant classes, attributes, and relationships for this problem.

Figure 8.1 Problem statement for the portfolio manager

- Discover classes by listing tentative classes and then rejecting spurious classes. [8.3.1–8.3.2]
- Discover associations by listing tentative associations, rejecting spurious associations, and refining the remaining associations. [8.3.3–8.3.5]
- Add tentative attributes of objects and links and then eliminate the spurious attributes. [8.3.6–8.3.7]
- Use generalization to note similarities and differences. [8.3.8]
- Test access paths. [8.3.9]
- Iterate and refine the model. Consider shifting the level of abstraction. [8.3.10–8.3.13]
- Organize the object model. [8.3.14]

8.3.1 Listing Tentative Classes

You should list tentative classes according to the following considerations:

- **Nouns and noun phrases**. Look for nouns and noun phrases in the problem statement. Figure 8.2 highlights the first occurrence. (We have omitted the examples shown in Figure 8.1.)

> Develop **software** for managing **investment portfolios**. The following **capabilities** must be provided:
> - Permit a **mixture** of **assets** and **liabilities**, including **bonds**, **stocks**, **options**, **commodities**, **mutual funds**, **precious metals**, **collectibles**, **cash**, **insurance**, **real estate**, and **loans**. A **financial instrument** is an asset, liability, or **portfolio**.
> - Support **hierarchies** of portfolios. Portfolios contain financial instruments.
> - Record **dates** of **purchase** and **sale** of financial instruments and their **price**. Allow for **stock splits** and **issuance** of different **kinds** of **securities**.
> - Track **value** of assets over **time**.
> - Handle **dividends**, both stock and cash. Handle **interest payments**.
> - Compute **ROI**.
> - Handle **currency conversions** with possible **fees**.
> - The **developer** should look at **broker statements**, **stock exchange listings**, existing **financial programs**, and other **sources** of **information** in order to discover relevant **classes**, **attributes**, and **relationships** for this **problem**.

Figure 8.2 Problem statement with highlighted nouns and noun phrases

- **Passive voice**. Watch for missing subjects in passive sentences. In the problem statement, the second sentence in the first paragraph has passive voice with an implicit subject of the "software developer." There is no other passive voice because the problem statement is worded as a directive to the software developer. Many problem statements are phrased this way.
- **Implicit classes**. Some classes are implicit or taken from general knowledge.

  The advisory statement at the end of the problem statement "the developer should look at. . ." accentuates some other sources of information. You must consider all available sources of information, as we described in Section 8.1.

  We add the following classes based on general knowledge: *FinancialInstitution* (the holder of portfolios) and *Person* (the owner of a portfolio).
- **Kinds of classes**. Model both physical entities (cash, precious metal) and conceptual entities (financial instrument, portfolio).
- **Reuse**. When possible, reuse class definitions from previous development efforts. We are developing the portfolio manager as an example and have no related software with which to achieve reuse. However, a colleague may develop software for forecasting stock market prices. We would expect to share some class definitions between the portfolio manager and the forecast software.

- **Naming**. Carefully choose class names. Crisp names promote understanding and concurrence as to the precise scope of the software. For example, we decided to represent option with the class name *StockOption* for clarity. Don't be surprised by fervent arguments between proponents of alternative names. You should try to use vocabulary from the application and avoid inventing terms that confuse application experts.
- **Missing classes**. Missing classes eventually become apparent as a gap in access paths or a missing target or argument of an operation.

8.3.2 Eliminating Spurious Classes

After listing the classes, you should review them to eliminate unnecessary and incorrect ones. Use the following criteria. (The last bullet in Figure 8.2 is really just admonishing the developer and has no description of requirements. Thus we disregard it in the following list.)

- **Redundant classes**. Do not keep multiple classes that express the same information. Choose the most descriptive and precise name and systematically use this name. For the portfolio manager, *InvestmentPortfolio*, *Mixture*, and *Portfolio* are redundant; we retain *Portfolio*. We eliminate *Security* and keep *Asset*. We discard *CurrencyConversion* as being redundant with *Purchase* and *Sale*.

  We decide to combine the notions of *Asset* and *Liability*. A liability really is an asset with a negative quantity. Then a positive quantity of stock is owned stock; a negative quantity is borrowed stock. This combination yields a nice symmetry; any asset can be bought or sold, owned or borrowed. Because we are combining *Asset* and *Liability*, we should also combine *Bond* and *Loan*. A bond is money owned; a loan is money borrowed.
- **Irrelevant or vague classes**. Every class should be clearly defined and necessary. This requires some judgment because the relevance of a class is dictated by the purpose of an application. For the portfolio manager, the following tentative classes are vague and should be discarded: *Software*, *Capabilities*, *Issuance*, and *Kinds*.
- **Attributes**. Restate as attributes tentative classes that are used only as values. If the independent existence of a property is important, make it a class and not an attribute. We should model *date*, *price*, and *value* as attributes for the portfolio manager. *Fee* also is an attribute and a consequence of purchases and sales.
- **Operations**. If an operation is manipulated as an entity, it is a class. Classes describe objects that have identity, may form relationships to other objects, are described with attributes, and bear functionality. If an operation fails these tests, eliminate it as a class.

  ROI is an operation to be provided by the portfolio manager software. The computation of return on investment will be complex, involving discounted cash flow to reflect the temporal value of money.
- **Roles**. The name of a class should reflect its intrinsic nature and not a role that it plays in an association. We clarify our portfolio manager model by adding roles. *FinancialInstitution* assumes the role of *accountHolder*. *Person* assumes the role of *accountOwner*. Various roles for persons often arise in applications.

- **Derived classes**. During analysis, avoid derived classes unless they appear in the application description. You should try to defer derived classes until design. The portfolio manager problem statement mentions stock dividends. For now we will treat each stock dividend as a combination of a cash dividend and a purchase.
- **Implementation information**. During analysis, avoid implementation details and concentrate on essential information. All classes must make sense for the application; avoid implementation constructs, such as linked lists and trees.

  We discard *Hierarchy*. Various implementation approaches can all yield the effect of hierarchies of portfolios. The purpose of analysis is to capture the essence of a problem, not a solution approach. We cannot emphasize this point enough. We have encountered programming books that would stress reuse from a *Tree* class in the class library at this point in the problem understanding. This is shortsighted; resist the urge to commit prematurely to an implementation approach until you fully grasp the problem. The key observation during analysis is that the portfolio manager involves recursion and the approach for realizing recursion is not yet decided.

  The problem statement refers to time; the composition and value of a portfolio change over time. During analysis, it is not apparent how we will handle temporal information in the final system. We will postpone the issue of temporal data until the design phase (Chapter 9). Our analysis model will deal only with the current value of portfolios.

We know little about commodities and have arbitrarily decided to eliminate them from further consideration. However, we will make sure the architecture readily extends to additional investment types, such as commodities. Such a revision to the problem statement is common. In most cases, the problem statement is not an absolute statement of requirements. Requestors can make mistakes and may reconsider.

The following valid classes remain at this stage in our analysis: *FinancialInstitution*, *Person*, *Asset*, *Bond*, *Stock*, *StockOption*, *MutualFund*, *PreciousMetal*, *Collectible*, *Cash*, *Insurance*, *RealEstate*, *FinancialInstrument*, *Portfolio*, *Purchase*, *Sale*, *Dividend*, and *InterestPayment*. We have observed proper OMT protocol for class names, restating plural nouns in the singular form.

For now we retain an additional class that we will revisit later in analysis. The notion of a *StockSplit* is relevant, but we must broaden the concept to accommodate the examples in Figure 8.1. (We also defer consideration of time until the system design phase.)

8.3.3 Listing Tentative Associations

You should list tentative associations found using the following guidelines:

- **Verb and prepositional phrases**. Look for verb phrases and prepositional phrases in the problem statement. These phrases follow the pattern *noun verb noun* or *noun preposition noun*. Figure 8.3 highlights candidate phrases.

  Note that although it has a noun verb noun construction, *a financial instrument is an asset, liability, or portfolio* is not a candidate association. Rather it is an indicator of generalization (an *is-a* phrase; see Section 8.3.8).

Develop **software for managing investment portfolios**. The following capabilities must be provided:

- Permit a **mixture of assets and liabilities**, including bonds, stocks, options, commodities, mutual funds, precious metals, collectibles, cash, insurance, real estate, and loans. A financial instrument is an asset, liability, or portfolio.
- Support **hierarchies of portfolios**. **Portfolios contain financial instruments**.
- Record dates of **purchase and sale of financial instruments** and their price. Allow for stock splits and **issuance of different kinds of securities**.
- Track **value of assets** over time.
- Handle dividends, both stock and cash. Handle interest payments.
- Compute ROI.
- Handle **currency conversions with possible fees**.

Figure 8.3 Problem statement with highlighted verb and prepositional phrases

- **Dependencies**. A dependency between two or more classes may be an association. The problem statement for the portfolio manager expresses dependencies rather vaguely. For example, the system must record *purchase and sale of financial instruments*. Many problems have more explicit statements of dependency, such as *person works for company* and *flights arrive and depart from airports*. Regardless, note any dependencies you can glean from the problem statement, because they represent possible associations.
- **Implicit associations**. Some associations are implicit. For example, *a financial institution holds portfolios* and *persons own portfolios*.
- **General knowledge**. Other associations are taken from general knowledge. For example, *a bond has interest payments*, *a stock has dividends*, and *a stock option has an underlying stock*.
- **Kinds of associations**. Model various kinds of associations, including physical location (*next to*, *contained in*), directed actions (*drives*), communication (*talks to*), ownership (*has*, *part of*), or satisfaction of some condition (*works for*, *married to*, *manages*). For example, *portfolios contain financial instruments*.
- **Avoid pointers**. Do not bury object references as pointers or pseudo attributes. There are different ways to implement associations, but these are considerations for design, not analysis. For example, *Stock* is not an attribute of *Dividend*; it is a class with which *Dividend* is associated.

Figure 8.4 summarizes possible associations for further consideration.

8.3.4 Eliminating Spurious Associations

The next task is to eliminate unnecessary and incorrect associations using the following criteria:

Verbs and prepositional phrases:
- software for managing investment portfolios
- mixture of assets and liabilities
- hierarchy of portfolios
- portfolios contain financial instruments
- purchase and sale of financial instruments
- issuance of different kinds of securities
- value of assets
- currency conversions with possible fees

Implicit verb phrases:
- a financial institution holds portfolios
- persons own portfolios

Knowledge of problem:
- a bond has interest payments
- a stock has dividends
- a stock option has an underlying stock

Figure 8.4 Tentative associations for the portfolio manager

- **Associations between eliminated classes**. If you have eliminated one of the classes in an association, you must eliminate the association or restate it in terms of other classes. For the portfolio manager we can eliminate the following candidate associations: *software for managing investment portfolios*, *hierarchy of portfolios*, *value of assets*, and *currency conversions with possible fees*.
- **Irrelevant or implementation associations**. Eliminate any associations that are outside the problem domain or deal with implementation constructs. For example, we do not want to be induced to make premature implementation decisions by the phrase *hierarchy of portfolios*.
- **Actions**. An association should describe a structural property of the application, not a transient event. The portfolio manager has no actions that are mislabeled as associations. We retain the clause *purchase and sale of financial instruments*. In one sense purchase and sale are events that occur, but we need to record descriptions of these events. We must track date of purchase or sale, price, and profit or loss.
- **Ternary associations**. You can decompose most associations between three or more classes into binary associations or phrase them as qualified associations. We can restate the phrase *mixture of assets and liabilities* as *mixture of assets* (because we have discarded liabilities as a class). This is a rather trivial decomposition. You will often encounter phrases that follow the pattern *noun verb noun preposition noun* (such as *person works-for division of company*). Decompose them into their binary constituents.

- **Derived associations**. Omit derived associations or explicitly mark them. None of the tentative associations for the portfolio manager appear to be derived. Later in our analysis we will encounter derived information. For example, portfolios have values and profits or losses that are systematically computed from assets and lesser portfolios.

We again defer consideration of complex situations, such as stock splits. Thus we postpone analysis of the following clause: *issuance of different kinds of securities*. Figure 8.5 shows the associations remaining at this stage of analysis. (We have substituted preferred class names for synonyms.)

- portfolio of assets
- portfolios contain financial instruments
- purchase of financial instruments
- sale of financial instruments
- a financial institution holds portfolios
- persons own portfolios
- a bond has interest payments
- a stock has dividends
- a stock option has an underlying stock

Figure 8.5 Remaining associations for the portfolio manager

8.3.5 Refining Associations

The last activity in discovering associations is to refine those that remain. You should further specify the semantics of associations using the following criteria:

- **Premature implementation**. Be careful to represent the logical intent, not an implementation approach. We have seen some developers of relational database applications become confused by associations. These developers try to express their thoughts directly with tables and bury associations as pointers. In reality, there are many ways to implement associations. By thinking in terms of tables, these developers violate the precepts of software engineering—limiting their conceptual ability, obscuring communication with others, and prematurely making detailed decisions before a broad understanding is in place.
- **Misnamed associations**. Choose names carefully. Don't say how or why a situation came about, say what it is. Names are important to understanding, and you should choose them with great care. You may omit association names when there is no ambiguity. Restate the awkward phrase *portfolio of assets* as *portfolio contains assets*.
- **Role names**. Add role names that appear in the problem statement. Also consider adding role names to distinguish multiple associations between classes. Role names are especially important for reflexive associations (see Section 2.2.3). In the association be-

tween *Stock* and *StockOption*, we can refer to *Stock* with the role of *underlyingSecurity*. Each person who owns a portfolio assumes the role of *accountOwner*; financial institutions assume the role of *accountHolder*.

- **Qualifiers**. Use qualifiers where possible. Qualifiers often arise from the need to specify a name within some context; most names are not globally unique. You will occasionally encounter qualification cascades (see Section 3.2.5), in which an accumulation of names denotes an object. Qualifiers are not apparent at this step of the portfolio manager analysis. They will appear later.
- **Multiplicity**. Specify multiplicity for the associations in your model, but don't dwell on it. The multiplicity often changes as your model evolves. You should always challenge multiplicity values of "exactly one." Usually the multiplicity is "many" (zero or more) or "zero or one." Similarly, you should challenge multiplicity values of "many." Sometimes you can restate a multiplicity of "many" with a qualifier or with a designation of *ordered*.

  We make the following multiplicity adjustments based on our understanding of the problem. We allow assets to belong to multiple portfolios. A portfolio may have multiple owners (such as a husband and wife) and need not be held by an institution (maybe the portfolio is hypothetical and exists only to facilitate ROI calculations). A stock may have many stock options.
- **Association classes**. You may have a class that would be better stated as an association class. An association class can participate in associations, just like an ordinary class. The difference is that an association class derives identity from its constituent classes, while a class has intrinsic identity (see Section 2.2.5). The portfolio manager does not have any association classes.
- **Missing associations**. Add missing associations. You can often discover these by testing access paths in your object model, especially during construction of the functional model.
- **Aggregations**. Sometimes it is difficult to distinguish aggregation from association. Do not worry if this occurs; for these cases the distinction is probably unimportant. When in doubt, use ordinary association. For bill-of-material problems, you should carefully consider whether to use physical aggregation or catalog aggregation (see Section 3.3.1).

  The word *contain* in the association *portfolios contain financial instruments* is highly suggestive of aggregation. Similarly, note the phrase *portfolio contains assets* (see the "Misnamed associations" bullet).

Figure 8.6 presents our initial class diagram for the portfolio manager.

8.3.6 Listing Tentative Attributes of Objects and Links

Once you have established the basic structure of your object model, you should elaborate it by describing the classes and associations. List tentative attributes according to the following criteria:

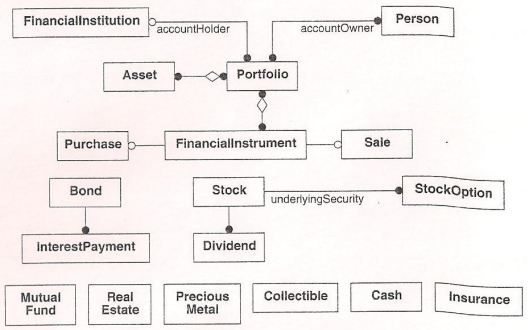


Figure 8.6 Initial class diagram for the portfolio manager

- **Possessive phrases and enumerations**. Look for nouns followed by possessive phrases in the problem statement, such as *the population of the city* or *the password of the user*. Adjectives often represent specific enumerated values, such as *red*, *green*, or *blue*. Figure 8.7 highlights candidate phrases in the problem statement.

> Develop software for managing investment portfolios. The following capabilities must be provided:
> - Permit a mixture of assets and liabilities, including bonds, stocks, options, commodities, mutual funds, precious metals, collectibles, cash, insurance, real estate, and loans. A financial instrument is an asset, liability, or portfolio.
> - Support hierarchies of portfolios. Portfolios contain financial instruments.
> - Record **dates of purchase and sale** of financial instruments and **their price**. Allow for stock splits and issuance of different kinds of securities.
> - Track **value of assets** over time.
> - Handle dividends, both stock and cash. Handle interest payments.
> - Compute ROI.
> - Handle currency conversions with possible fees.

Figure 8.7 Problem statement with highlighted attribute phrases

- **Missing attributes**. Unlike classes and associations, attributes are less likely to be described fully in the problem statement. You must draw on your knowledge of the application and expert advice to find them. Fortunately, attributes seldom affect the basic structure of a problem. For the portfolio manager, we know that there is preferred stock and common stock, so we add the *isPreferred* attribute to *Stock*.
- **Limit discovery of attributes**. Do not carry discovery of attributes to excess. Consider only attributes that directly relate to the application being developed.
- **Avoid pointers**. Attributes should not refer to objects; use associations to describe dependencies between objects.

8.3.7 Eliminating Spurious Attributes

You should eliminate unnecessary and incorrect attributes using the following criteria:

- **Objects**. If the independent existence of an entity is important, rather than just its value, it is an object. Scrutinize the classes and tentative attributes to make sure you have not mislabeled any attributes or classes.
- **Qualifiers**. If the value of an attribute depends on a particular context, consider restating the attribute as a qualifier. Names are often better modeled as qualifiers rather than object attributes. Qualifiers are not yet apparent for the portfolio manager problem.
- **Identifiers**. Object-oriented languages and object-oriented databases incorporate the notion of an object identifier for unambiguously referencing an object. Do not list these object identifiers, because they are implicit in object models. List only attributes that exist in the application.
- **Link attributes**. If a property depends on the presence of a link, it is an attribute of the link, not of a related object. Link attributes are obvious for many-to-many associations; you cannot ascribe them to either class because of the multiplicity. Link attributes also arise for one-to-many and one-to-one associations. Assign each attribute to the class or association it describes most directly and naturally.

  For the portfolio manager *quantity* is a link attribute of the association between *Asset* and *Portfolio*. We could also allow for unequal ownership of portfolios. The association between *Person* and *Portfolio* would then have the link attribute *fractionalOwnership*. We will not add this complication.
- **Fine detail**. Omit minor attributes that are not important for the application.
- **Derived attributes**. Omit or clearly label derived attributes.
- **Discordant attributes**. An attribute that seems completely different from and unrelated to all other attributes may indicate a class that should be split into two distinct classes. A class should be simple and coherent. If you encounter many discordant attributes, your analysis is likely flawed. By its very nature an object-oriented approach is centered about things and descriptions of things, so awkward combinations should seldom arise.

- **Boolean attributes**. Reconsider all boolean attributes. Often (but not always) you can broaden a boolean attribute and restate it as an enumeration [Coad-95].

  For the portfolio manager we will restate the *isPreferred* attribute of *Stock* more broadly as *stockType* with the values *common*, *preferred*, *closed-end*, and *other*.

After applying these criteria to the portfolio manager model, we get the diagram in Figure 8.8. We continue to defer the handling of stock splits. For simplicity, we assume that stocks have a single ticker symbol used on all stock exchanges. Attribute *portfolioType* characterizes portfolios for tax purposes; portfolio types of *ordinary*, *IRA*, and *401K* would be appropriate for the United States.

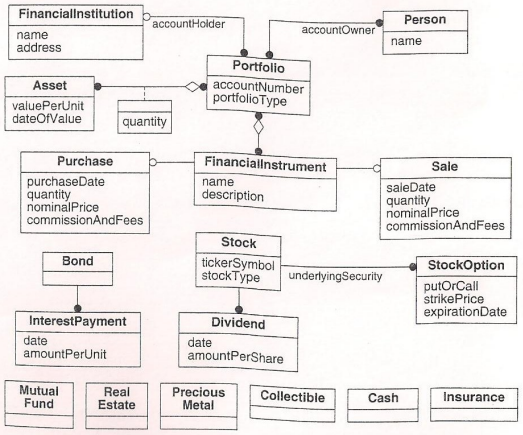


Figure 8.8 Class diagram with attributes for the portfolio manager

8.3.8 Using Generalization to Note Similarities and Differences

The next step in object modeling is to organize classes using inheritance. You can add inheritance in one of two directions: top down or bottom up.

- **Top-down generalization**. Specialize existing classes into subclasses. Sometimes there is an obvious need to partition. Look for lists of special subcases in the problem statement. Also look for noun phrases composed of various adjectives on a class name. Often you can think of additional special cases. Unify any redundant attributes that now become apparent. You should avoid creating unnecessary generalizations for enumeration attributes. Specialize a class only when the subclasses have distinct attributes, associations, methods, and/or semantic behavior.

  The first bullet of the portfolio manager problem statement (see Figure 8.1) lists several forms of assets (the "including" clause). If these examples were of minor importance and were not described by attributes, associations, or operations, they would merely be enumeration values. However, different kinds of investments have distinctive behavior. Thus we will treat *Bond*, *Stock*, *StockOption*, *MutualFund*, *PreciousMetal*, *Collectible*, *Cash*, *Insurance*, and *RealEstate* as subclasses of *Asset*. (Recall that in Section 8.3.2 we discarded *Commodity* as a minor refinement of the problem statement. We also unified the notions of *Bond* and *Loan*.)

  The second sentence in the first bullet states that *a financial instrument is an asset, liability, or portfolio*. The *is-a* phrase is an obvious indicator of generalization. (Once again there are only two subclasses because we have combined *Asset* and *Liability*.)

- **Bottom-up generalization**. Find classes with similar attributes, associations, or operations. Create an abstract class to share common information. You may have to alter the definitions a bit; this is acceptable as long as generalization semantically applies. Symmetry will suggest missing classes.

  In Figure 8.8 a bond may have many interest payments. However, a mutual fund comprised of bonds also may pay interest. Furthermore, both a stock and a mutual fund of stocks may pay dividends. We decide to split *MutualFund* into *BondFund* and *StockFund*. Then we generalize *Bond* and *BondFund* into *BondAsset*, which pays interest. We also generalize *Stock* and *StockFund* into *StockAsset*, which pays dividends.

  In principle, a fund could combine bonds and stocks. However, most funds are more focused to suit investor taste. We could always add mixed funds to the portfolio manager in the future if needed. This process of defining the precise boundary for a system often occurs in practice. The software requestor and the developer must decide what is within the scope of a system and what is not. A major virtue of object-oriented modeling is that the scope is made clearly visible. You can readily extend a carefully modeled and architected system.

  We do not show the composition of a stock fund in terms of its underlying stocks. Similarly, we do not show the composition of a bond fund. You should only model information that is important to an application. Stock funds and bond funds are assets and therefore have a direct market value. We do not need the internal structure of stock funds and bond funds for computing portfolio value and ROI.

  At this point we notice that *Stock* and *StockFund* have the same attributes, associations, methods, and behavior. Hence there is no need to show the subclasses; we can merely show *StockAsset* in the object model. Similarly, the distinction between *Bond* and *BondFund* is inconsequential and we can simply show *BondAsset*.

  We could also choose to regard stock option as a stock asset, and we considered this at first. However, stock options do not pay dividends, and we decided that the similarity between stock and stock fund was much more significant than the commonality with stock option. We avoid this further generalization.

- **Adjust placement of associations**. As you introduce inheritance, you may need to adjust the targets of some associations. Move associations up or down the inheritance hierarchy as semantically appropriate. Symmetry may suggest additional attributes to distinguish among subclasses more clearly. You may need to decompose large classes so you can find similar components.

  We define an asset as something that has value and can be *directly* bought or sold. Thus a portfolio is not an asset; a portfolio of financial instruments cannot be directly bought and sold. Therefore, *Purchase* and *Sale* do not relate to a *FinancialInstrument* but to an *Asset*. Also, we shift the aggregation from *Asset* and *Portfolio* to *FinancialInstrument* and *Portfolio*; a portfolio contains financial instruments that can be assets or lesser portfolios.

- **Multiple inheritance**. You may use multiple inheritance to increase sharing, but only if necessary, because it increases conceptual complexity. Use of multiple inheritance is more appropriate during design, especially if you mixin functionality from a class library. (See Section 3.5.2.)

Figure 8.9 shows the portfolio manager object model after adding inheritance.

8.3.9 Testing Access Paths

You cannot fully verify access paths until you have constructed the functional model. Nevertheless, it is worthwhile to peruse the object model quickly for obvious omissions. Later, during construction of the functional model, we will check more thoroughly.

Trace access paths through the object model diagram to see if they yield sensible results. Where a unique value is expected, is there a path yielding a unique result? For multiplicity "many," is there a way to pick out unique values when needed? You may need to introduce some qualifiers. Are there useful questions you cannot answer?

The *accountNumber* really is a qualifier on the association between *Portfolio* and *FinancialInstitution*. The financial institution assigns the account number; an account number is meaningful only within the context of a financial institution. Account numbers uniquely identify a portfolio for a financial institution. *InterestPayment* and *Dividend* are qualified by *date*.

It is perfectly acceptable to have classes that are "disconnected" from other classes. This usually occurs when the relationship between a disconnected class and the remainder of the model is diffuse. The portfolio manager as presented here has no such disconnected classes. However, a prior draft had an *InflationIndex* class for adjusting ROI calculations for inflation. *InflationIndex* was stand-alone and had no relationships to other classes. (We cut the mention of inflation to simplify the case study.)

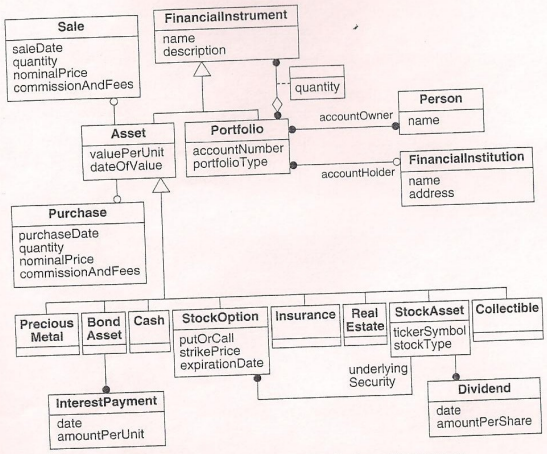


Figure 8.9 Class diagram with inheritance for the portfolio manager

8.3.10 Iterating and Refining the Model

A model is rarely correct after a single pass; our models are seldom correct on the first try. Most object models take several passes, interleaved with refinement of the dynamic and functional models. Actively seek the advice and criticism of others. You can improve a model by reconciling different perspectives.

We have found that revision often improves the ability of a model structurally to enforce application constraints. Qualifiers and association classes are especially important because they let you specify identity and navigation paths precisely. However, do not carry the enforcement of constraints to excess and distort your model. Elegance is also an important goal.

Look for patterns (see Section 4.3) as you refine your object model. You should be alert to patterns for several reasons. First, if you recognize a pattern, you can ensure that you use it correctly and fully understand the implications. If you draw your models so that a pattern

can be easily recognized, other persons will better understand your models. In Figure 8.9 the relationship between *FinancialInstrument*, *Asset*, and *Portfolio* instantiates the complex pattern for a directed graph (Figure 4.9).

You may wish to rework your object model to eliminate multiple inheritance. Multiple inheritance may not be worth the nuisance of explaining to your reviewers. Substituting a workaround for multiple inheritance may simplify the explanation of your model. In general, we are reluctant to use multiple inheritance in our analysis models; we are much more likely to introduce multiple inheritance during design for mixins, such as inheriting persistence from the class library of an object-oriented DBMS.

As you refine your object model, it is worthwhile to revisit names. Reviewers can often suggest better names with crisper meaning. Be wary of names that seem fine but have deleterious connotations within an organization. Reviewers will become confused by such names and draw erroneous conclusions, so you must revise them.

Next we address the requirements we have postponed since Section 8.3.2, building on the nucleus of our object model.

8.3.11 Shifting the Level of Abstraction

So far in the OMT analysis process, we have taken the problem statement quite literally. We have regarded nouns and verbs in the problem description as direct analogs of classes and associations. This is a good way to begin analysis, but it does not always suffice. Sometimes you must shift the level of abstraction to solve a problem. On occasion it is helpful to introduce more abstraction with metamodels, generic classes, and reification. (See Chapter 4.)

For example, we have developed several bill-of-material applications. (See Section 3.3.) For several reasons, it is often difficult to model precisely the parts explosion. Our understanding of the problem domain is incomplete. Furthermore, different organizations have different opinions about the appropriate decomposition. The manufacturing staff believe a bill-of-material should be organized by the sequence of manufacturing steps. In contrast, engineers prefer to organize a bill-of-material on the basis of functionality. And there are additional viewpoints. We have found that the preferred organization varies from company to company and can change over time.

For some applications our solution has been to elevate the abstraction of the model and shift toward a generic model of parts (Figure 3.17) rather than a tangible model that directly corresponds to the application (Figure 3.18). Then our bill-of-material software straddles the argument about how best to decompose a bill-of-material. At run-time our software stores both the part definition and part instantiation, and we can change the structure of the bill-of-material without rewriting our software.

The bill-of-material example highlights another important philosophical point. Developers should try to decouple software from business decisions. Software should enable strategic business decisions, not dictate business decisions. Business leaders should be able to make decisions without inhibition from software systems. For our bill-of-material applications we have straddled the business decision of how best to organize the bill-of-material. We wrote software in a manner that is neutral with regard to the eventual decision, and it can tolerate various and possibly changing decisions.

Our first revision of the object model for the portfolio manager is motivated by a shift in abstraction.

8.3.12 First Revision: Introduce the Notion of a Transaction

We need to represent more abstractly the notion of purchase and sale. *Purchase* and *Sale* have similar attributes and differ only with regard to which party receives the proceeds. We introduce the notion of a transaction that may be a purchase or a sale of an asset. We must specify the portfolio that is the source of the assets that are sold or the portfolio that is the recipient of the assets that are purchased. Since *Purchase* and *Sale* have the same attributes, associations, and behavior, we need not represent them as subclasses and can instead add an enumeration attribute to *Transaction* indicating purchase or sale.

This revision corrects a flaw in our earlier model (Figure 8.9). Previously we indicated that each asset has at most one sale and at most one purchase. Then it becomes awkward to represent the situation in which someone purchases 500 shares of a stock and then sells 200 shares on one date and 300 shares on another date. More abstractly representing an asset as having many transactions resolves this problem.

Next we broaden our definition of a transaction. Each transaction may have a type of *purchase*, *sale*, *deposit*, *withdrawal*, *barter*, or *journal*. A *purchase* occurs when a person gives cash and receives an asset. A *sale* occurs when a person gives an asset and receives cash. A *deposit* is the simple addition of an asset to a portfolio. A *withdrawal* is the simple removal of an asset from a portfolio. *Barter* accommodates complex situations, such as stock splits and the issuance of different kinds of securities. A *journal* entry occurs when there is a discrepancy in the portfolio manager records that we (as users) wish to adjust manually. In reality, we can deduce *transactionTypes* other than *journal* from the links to *Asset*; however, we retain *transactionType* for convenience.

The *Transaction* class reifies trading actions that occur against a portfolio. We reify transactions so that we can record past financial activity. We must store both *transactionDate* (when the transaction occurs) and *recordDate* (when the transaction is recorded). The notion of a transaction is a general concept that arises with many other kinds of applications. This increases our confidence that the abstraction is correct.

A consequence of recording transactions is that the assets in a portfolio are derived. A person must define a portfolio, supplying a name, description, and portfolio type. A person must also specify which portfolios are contained in other portfolios. However, software can reprocess the log of transactions to determine the assets in a portfolio. The transaction is the *only* mechanism for changing the assets in a portfolio. We must now replace the association between *FinancialInstrument* and *Portfolio* by two associations—from *Portfolio* to *Asset* and *Portfolio* to *Portfolio*. Figure 8.10 illustrates that the association from *Portfolio* to *Asset* is derived; the association from *Portfolio* to *Portfolio* is not derived. We simplify the model by restricting portfolio composition to a collection of trees; a portfolio has at most one parent.

We can treat currency exchanges (such as exchanging U.S. dollars for Deutschemarks) as *purchase* or *sale* transactions involving two cash assets. For example, we can treat a sale of Deutschemarks as a transaction in which Deutschemarks are the asset (cash asset) that is

given and U.S. dollars are received. We treat expired options as a *sale* with receipt of zero cash. We model exercised options as a *barter*.

In Figure 8.10 we define *quantity* as positive if an asset is received in a transaction and negative if an asset is given. The *Asset* and *Portfolio* classes appear twice in the diagram for ease of layout. Our normal convention is to show the attributes (and possibly operations) for a class in one location where the class is "defined" and show the class icon (attributes and operations suppressed) for all other references to the class.

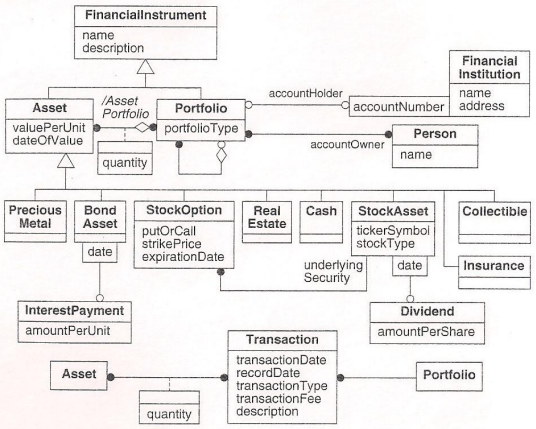


Figure 8.10 Class diagram revised to include transactions for the portfolio manager

8.3.13 Second Iteration: Analyze Existing Forms and Refine Transaction

After inspecting some financial statements, we found an oversight in the object model. Many bonds have a single return of principal; a bond fund may have periodic partial returns of principal. Consequently, we would add *ReturnOfPrincipal* to the object model.

The inspection also prompted us to extend our definition of a transaction to include *dividend*, *interest*, and *return of principal*. This extension eliminates several classes and lets us

directly handle stock dividends. We no longer need to treat a stock dividend as a cash dividend and a purchase (see the "Derived class" bullet in Section 8.3.2). The change clarifies the processing of dividends, interest, and return of principal; we intend that a transaction be the only means of changing the assets in a portfolio. We add an association to the *Transaction* class denoting the *assetReferenced*. For a dividend, the *assetReferenced* is a stock. For interest and return of principal, the *assetReferenced* is a bond. For all other transaction types, *assetReferenced* is null.

Figure 8.11 shows the final result of object modeling.

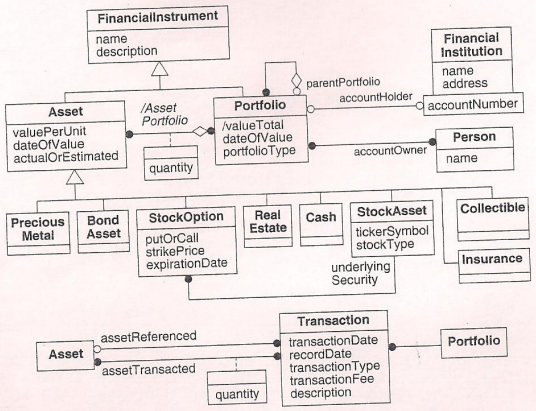


Figure 8.11 Final class diagram for the portfolio manager after object modeling

In practice, model building is not as rigidly ordered as we have shown. As you become proficient, you can perform the modeling steps in an alternative order. For example, for an experienced modeler the generalization of precious metals, bonds, cash, stocks, and so forth into an asset is apparent from the initial reading of the problem statement. Also, the generalization of *Asset* and *Portfolio* into *FinancialInstrument* is readily noted. You may infuse such obvious top-down generalization early in the analysis process. However, you should defer the search for similarities and differences to achieve reuse (bottom-up generalization) until

you have determined associations and attributes. This will help you avoid choosing a basis for generalization prematurely, which can lead to an inferior model.

8.3.14 Organizing the Object Model

The last step in object modeling is to organize your model. Flat class diagrams suffice for small and medium-sized problems. However, it is difficult to understand large applications (applications with more than five pages of class diagrams) all at once, and you should organize them into packages. (See Section 3.6.) Large applications may require several tiers as you decompose packages; this cascade of packages forms a hierarchy.

Each package should encompass a logical group of classes (with corresponding associations and generalizations). Each package should have a coherent theme; look for dominant classes, dominant relationships, and major aspects of functionality. A class or association class may appear in multiple packages, but ordinarily we show each association and generalization only once. By convention, we show the attributes (and possibly operations) for a class where the class is defined and show the class icon (attributes and operations suppressed) for all other references to a class.

The portfolio manager, as described, is too small to require packages. We have had to simplify the problem for presentation. A more complete portfolio manager could have a package for each type of asset. For example, a stock package might include information for forecasting future stock market prices. A real estate package might track depreciation, capital improvements, rental income, taxes, and cash flow.

It is also worth devoting some effort to the aesthetics of your diagrams. Place important concepts so they are immediately noticed. For the portfolio manager, we made *FinancialInstrument*, *Asset*, *Portfolio*, and *Transaction* visually prominent. The layout of your diagrams should elicit symmetry. For example, we try to place subclasses at the same level in the portfolio manager diagram. Minimize crossings and bends for association and generalization lines. Try to place the generalization superclass above its subclasses.

8.4 THE DATA DICTIONARY

The data dictionary defines all modeling entities (classes, associations, attributes, operations, domains, and enumeration values) and explains the rationale for key modeling decisions. The data dictionary highlights important assumptions and restrictions. You should not define by example, but it is helpful to give examples of the definition. You should also avoid circular definitions.

As you prepare a data dictionary, it is a good time to check for adherence to naming standards. Naming standards can help coordinate the work of multiple persons and projects. You may want to use a prefix for classes and associations indicating their primary package. Sometimes we make attributes globally unique by giving them a prefix of their class or association name. Also, you may want to standardize suffixes for different kinds of data. For example, in the portfolio manager, we incorporate the phrase *date* in the names of attributes

that are dates. All enumerations incorporate the phrase *type* or are named by the enumeration values separated by *or*.

You should also enforce conventions for abbreviations. Otherwise you will have trouble searching the data dictionary and your models. For example, you may require that the phrase *description* either be fully spelled or abbreviated as *desc*.

We now present definitions for the portfolio manager, organized about classes.

- *Asset*—something that has value and can be *directly* bought or sold. An asset with a negative quantity in a portfolio is a liability. Assets vary in their liquidity, that is the period of time typically required to achieve a purchase or sale. Assets may belong to multiple portfolios. We record the value of each asset, the date of valuation, and all transactions by which an asset is acquired or divested. Asset values may be actual or estimated. The portfolio manager supports the following types of assets: precious metal, bond, stock, stock option, real estate, cash, collectible, and insurance.
- *BondAsset*—a right to receive periodic interest payments and one or more returns of principal. The value of the principal is fixed on the redemption date(s), unlike stocks, whose value varies according to corporate fortunes and the economy. Companies must pay interest on bonds before issuing dividends on stock. Thus corporate bond holders receive a more secure income stream in return for lesser participation in corporate fortunes. The term "bond" encompasses annuities, municipal bonds, federal bonds, corporate bonds, zero coupon bonds, certificates of deposit, and savings bonds. A loan is a bond that is held short (money owed).
- *Cash*—currency or coinage, such as U.S. dollars, Deutschemarks, or Japanese yen. The name of the financial instrument (via inheritance) denotes the country of issue.
- *Collectible*—an item of artistic or aesthetic merit, such as paintings, coins, stamps, and antiques. Many collectibles have value that can be measured with purchases and sales in the marketplace.
- *FinancialInstitution*—a business that holds portfolio accounts. Banks, insurance companies, and stock brokerages are examples of financial institutions.
- *FinancialInstrument*—an asset or a portfolio.
- *Insurance*—a guarantee of payment for a specified situation (usually a crisis or disaster) in return for periodic payment of premiums. For the portfolio manager, we treat insurance as a financial instrument only if it has a surrender value. For example, life insurance is a financial instrument if the insured pays more than the current cost of benefits and accrues a residual value.
- *Person*—the owner of portfolios. We exclude other kinds of persons, such as stock brokers and fund managers, who are peripheral to the portfolio manager.
- *Portfolio*—a collection of financial instruments that may evolve over time through transactions. A portfolio consists of financial instruments which can be assets and lesser portfolios. A portfolio may be owned by multiple persons and may be held by some financial institution. We permit various portfolio types (such as normal, 401K, and IRA). We compute ROI and record the value of portfolios.

- *PreciousMetal*—a metal that trades at a great premium compared with base metals. For example, gold, silver, and platinum are commonly regarded as precious metals, while copper, lead, and aluminum are considered base metals. Precious metals derive their high values from high demand relative to supply. Precious metals complement a portfolio because they are liquid and resistant to inflated supply (unlike currency, which governments can print at whim). Precious metals have the drawback of not generating cash flow.
- *RealEstate*—land and/or buildings. Real estate incurs taxes and investment for capital improvement and returns rent or the benefits of personal use. This case study superficially considers the impact of a home on net worth. We would need to enhance the portfolio manager to handle investment property.
- *StockAsset*—an ownership stake in a company. Holders of stock benefit from possible dividends and anticipated increases in corporate net worth reflected in higher future stock prices. For most kinds of stock, the maximum possible loss is limited to the amount of investment. Stocks trade on various exchanges throughout the world that establish current value. For simplicity, we assume that stocks have a single ticker symbol across all exchanges. There are different types of stock (such as common, preferred, closed-end stock funds, and special classes) with various combinations of advantages and restrictions. A stock may have corresponding stock options.
- *StockOption*—a call or put option. A call (put) option is a right to purchase (sell) a stock on or before an expiration date at a strike price. The holder of a stock option has the choice of exercising the option or waiting; this choice ends on the expiration date. The seller of a stock option is subject to the whims of the option holders.
- *Transaction*—a trade of one collection of assets for another collection of assets. Each transaction may have a type of *purchase*, *sale*, *deposit*, *withdrawal*, *barter*, *journal*, *dividend*, *interest*, and *return of principal*. A *purchase* occurs when a person gives cash and receives an asset. A *sale* occurs when a person gives an asset and receives cash. Purchase and sale transactions often involve fees. A *deposit* is the simple addition of an asset to a portfolio. A *withdrawal* is the simple removal of an asset from a portfolio. *Barter* accommodates complex situations, such as stock splits, stock option exercises, and issuance of different kinds of securities. A *journal* entry occurs when there is a discrepancy in the portfolio manager records that the user wishes to adjust manually.

  A *dividend* is a periodic payment of cash or stock to the holder of a stock. *Interest* is the periodic payment of cash to the holder of a bond. *Return of principal* is the payment of the principal (original cash tendered) for a bond. Bonds may return principal in one final payment or in multiple payments throughout the life of the bond.

  In reality, we can deduce most *transactionType* values from the links to *Asset*; however, we retain *transactionType* for convenience. Note that the *Transaction* class reifies trading actions that occur against a portfolio. The transaction is the *only* mechanism for changing the assets in a portfolio. We reify transactions so we can recall past financial activity. We note two kinds of dates for transactions: *transactionDate* (the date when the transaction occurs) and *recordDate* (the date when the transaction is recorded).

8.5 DYNAMIC MODEL

As we described earlier, the dynamic model characterizes the temporal interactions of objects and their responses to events. The first step in dynamic modeling is to construct a state diagram for each class with important temporal behavior—that is, a class with multiple states or multiple responses to stimuli. Other classes have degenerate state diagrams, and you need not show them. Some temporally important classes are ***active*** and control an application, such as the controller for a user interface; active objects determine the sequence of states for other objects. The remaining temporally important classes are ***passive*** and impose no control on other classes; passive objects have an interesting life history or constraints that depend on the current state. As an example of a passive state diagram, consider an order entry system; an order passes through well-defined states, and corporate policies may need to be enforced on a change in state. The collection of state diagrams constitutes the dynamic model.

The portfolio manager case study has a trivial dynamic model. The user interface is the only active class, and there are no passive classes. All the other classes of the portfolio manager problem have no change of state. Database applications in general tend to have simple dynamic models.

In principle, we could construct a state diagram for the user interface of the portfolio manager. But such a state diagram would be of little help. For many data management applications, you can build a user interface with a form and report builder, essentially a user interface framework. We used such a framework with our MS-Access implementation (see Chapter 14). The developers of a user interface framework have essentially already constructed a dynamic model for us. Further exploration of the user interface is outside the scope of this book.

If you are interested in learning more about the dynamic model, [Rumbaugh-91] gives a detailed explanation.

8.6 FUNCTIONAL MODEL

The functional model defines the operations that objects perform and to which they are subjected. Operations arise in the object model from queries, updates, derived data, and constraints. Operations arise in the dynamic model from events, guard conditions, actions, and activities. During analysis you should establish that a reasonable algorithm exists and try to defer the details of the algorithm until design.

We use multiple paradigms to express the functional model, including decision tables, mathematical equations, and enhanced pseudocode. The choice of appropriate paradigm depends on the particular application.

You should perform the following steps during functional modeling:

- Specify use cases. [8.6.1]
- Choose one or more functional modeling paradigms. [8.6.2]
- Specify operations from the object model. [8.6.3]

- Specify operations from the dynamic model. [8.6.4]
- Verify the entire analysis model. [8.6.5]

8.6.1 Specifying Use Cases

Once you have a sound object model, you should specify use cases. A ***use case*** [Jacobson-92] is a theme for interacting with a system. You should think of the various external entities that interact with the system and the different ways they each use the system. For the portfolio manager we can only think of two external entities: the user of the system and financial institutions that provide financial data and handle transactions. Use cases let you concisely list different kinds of behavior. Figure 8.12 lists use cases for the portfolio manager. It is by no means complete; you can always devise new ways to explore a database.

- Add, delete, and update a person.
- Add, delete, and update a financial institution.
- Add, delete, and update a financial instrument.
- Add and delete account owners.
- Add and delete a portfolio-to-portfolio link (adjust portfolio composition).
- Add, delete, and update a transaction.
- Add, delete, and update asset prices for various dates.
- Find the account number for a portfolio.
- Find all dividends for a stock.
- Find all interest payments and returns of principal for a bond.
- Find all transactions for an asset other than dividend, interest, and return of principal.
- Find all transactions for a portfolio.
- Calculate the total transaction fees incurred for a portfolio for an interval of time.
- Calculate the assets in a portfolio on a given date.
- Find the stock options in a portfolio that will expire one month from now.
- Calculate the value of a portfolio on a date.
- Calculate the ROI for a portfolio for an interval of time.

Figure 8.12 Use cases for the portfolio manager

Many developers prefer to let the specification of use cases drive the discovery of classes and associations during construction of the object model. This is a reasonable variation of the OMT process. In a sense this is what we actually did in our construction of the object model. The portfolio manager requirements embody the rudiments of use cases.

But our expression of use cases early in analysis would have been raw, because our understanding of the relevant objects and their relationships was incomplete. For example, at the beginning of analysis we were not aware of the importance of transactions. We discovered transactions as we worked through the process of constructing the object model. If you