

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.

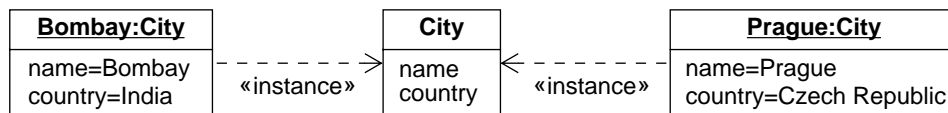


Figure 3.1 Instantiation relationships

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.

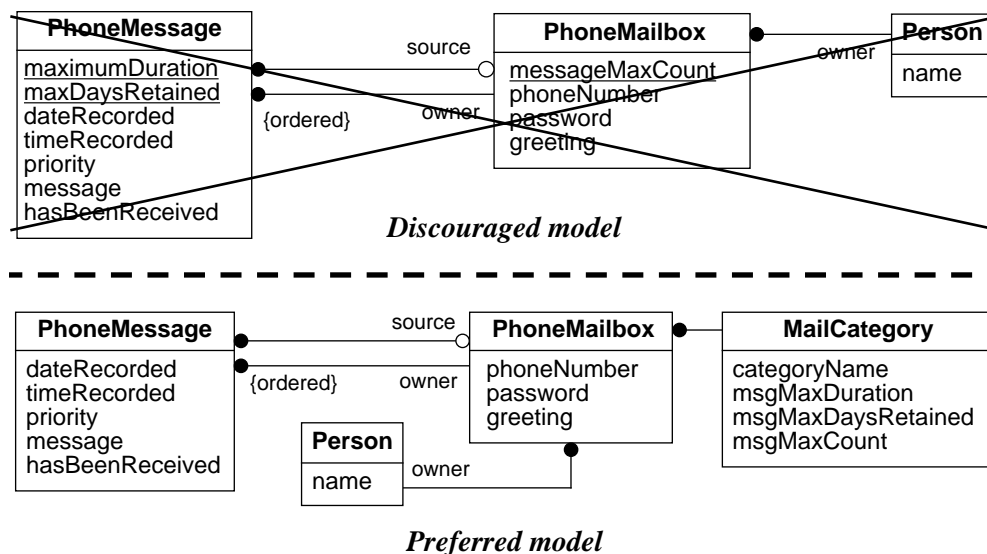


Figure 3.2 Instead of using class attributes, model groups explicitly

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.

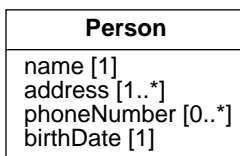


Figure 3.3 Attribute multiplicity

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.

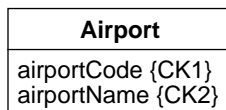


Figure 3.4 Candidate keys for a class

We indicate a candidate key for a class with the notation *CK_n* 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 *CK_n* 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.

PhoneMessage
dateRecorded:Date timeRecorded:Time priority:PriorityType=NORMAL message:LongString hasBeenReceived:Boolean=FALSE

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.

Person
name [1] : Name address [1..*] : Address street city state mailCode country phoneNumber [0..*] : PhoneNumber birthDate [1] : Date year month 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.

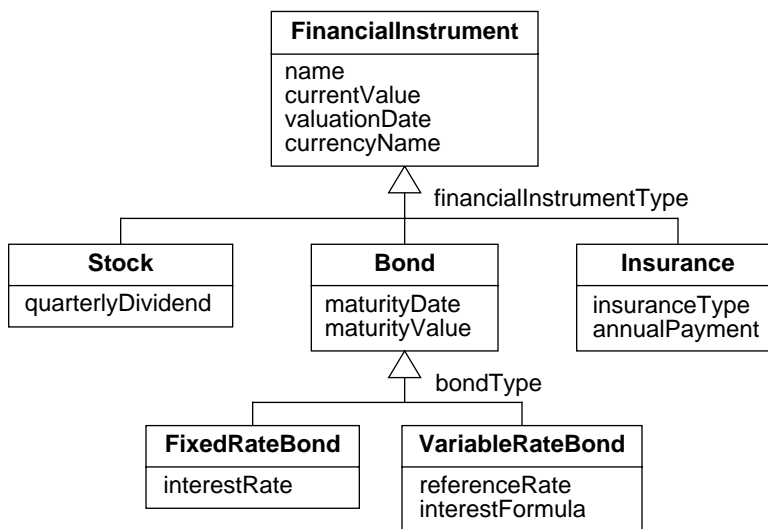


Figure 3.8 Secondary data for attribute values

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.

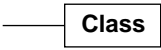

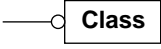


<i>OMT construct</i>	<i>Minimum multiplicity</i>	<i>Maximum multiplicity</i>
	<i>1</i>	<i>1</i>
	<i>0</i>	<i>infinite</i>
	<i>0</i>	<i>1</i>
	<i>1</i>	<i>infinite</i>
	<i>2</i>	<i>4</i>

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.

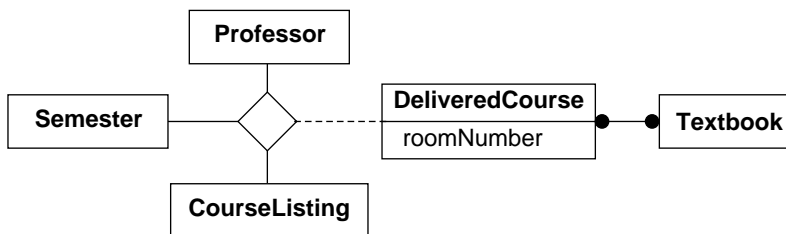
Maximum multiplicity is the upper limit on the possible number of related objects. The most common values are one and infinite.

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).

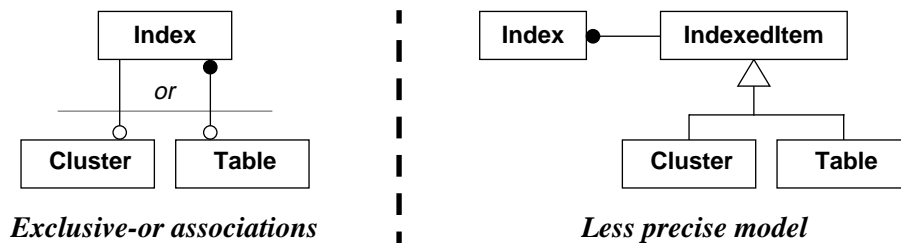


Figure 3.11 Exclusive-or associations can yield a more precise model

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.

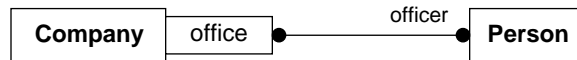


Figure 3.12 Qualification need not yield a target multiplicity of “one”

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.

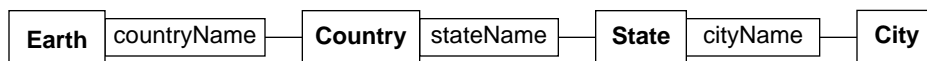


Figure 3.13 A qualification cascade denotes increasingly specific objects

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.

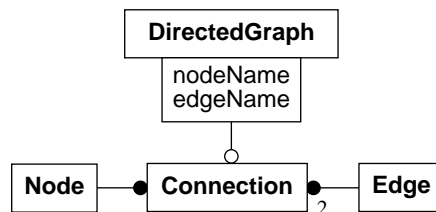


Figure 3.14 A compound qualifier refines the multiplicity of an association

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.

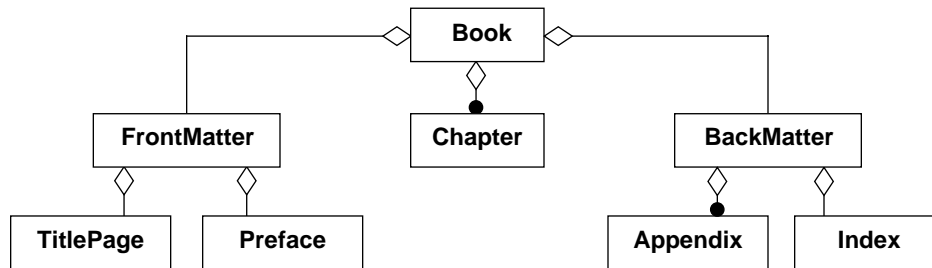


Figure 3.15 Aggregation is a kind of association with additional semantic properties

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 matter. 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.)

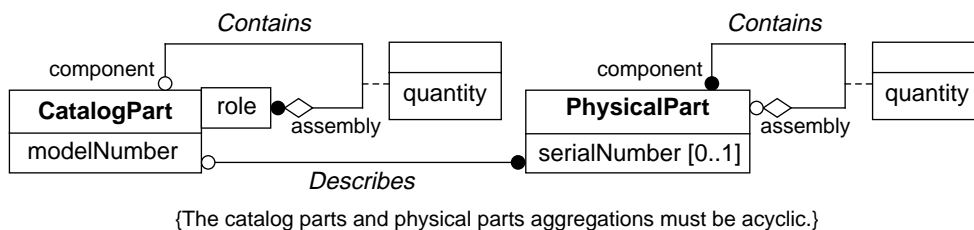


Figure 3.17 Relationship between catalog aggregation and physical aggregation

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.

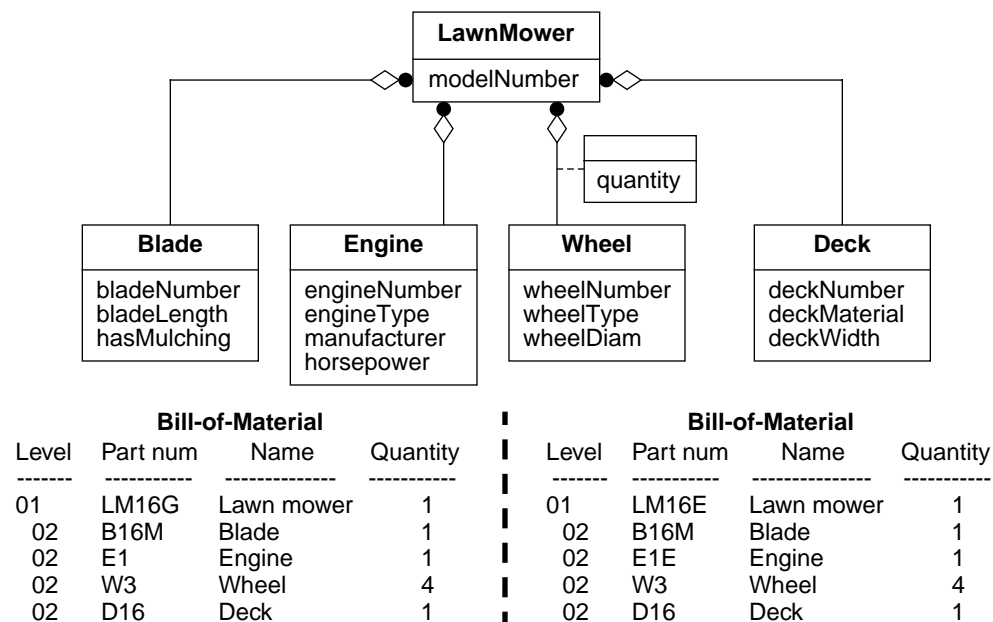


Figure 3.18 Structured part names for catalog aggregation

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.

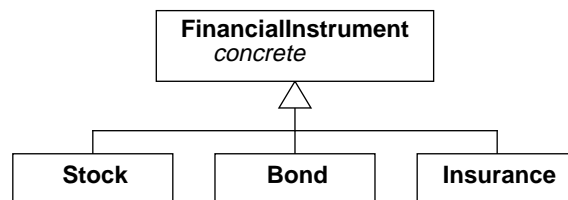


Figure 3.19 Concrete classes

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.

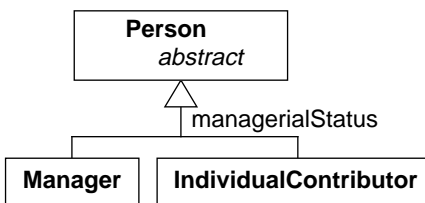


Figure 3.20 An abstract class

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.

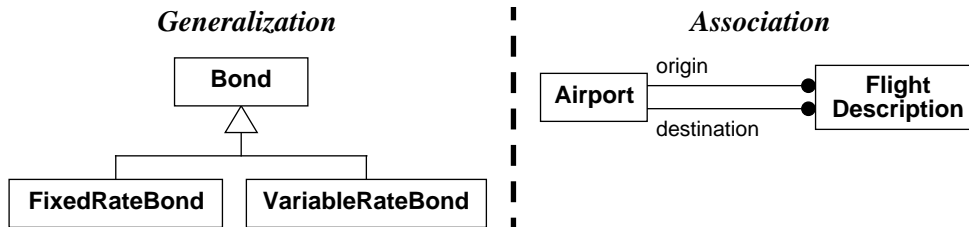


Figure 3.21 Generalization versus association

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.

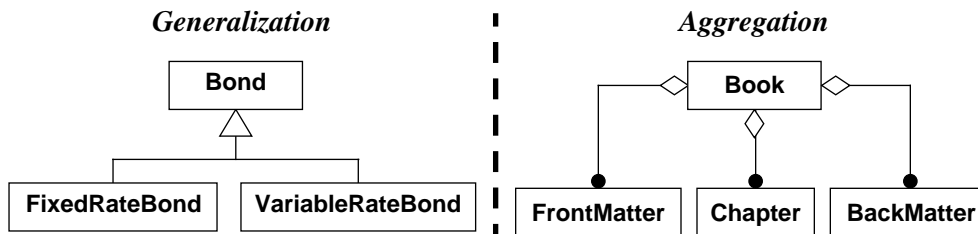


Figure 3.22 Generalization versus aggregation

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

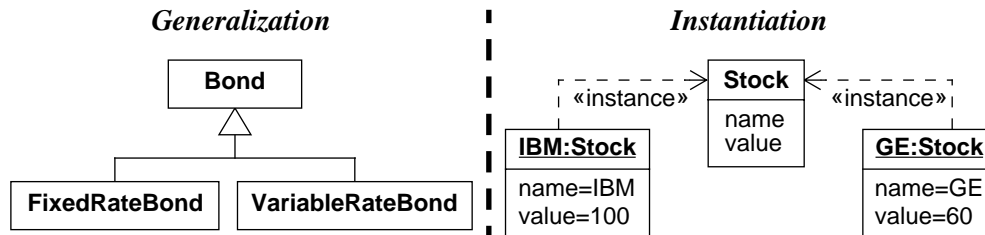


Figure 3.23 Generalization versus instantiation

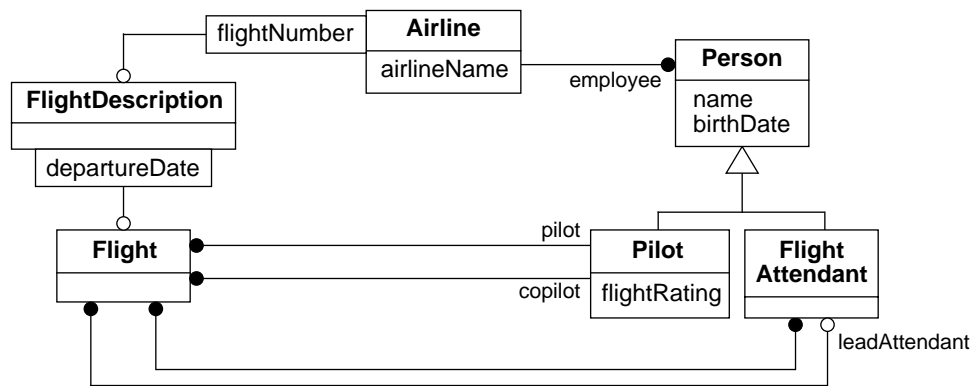


Figure 3.24 Subclass versus role

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-

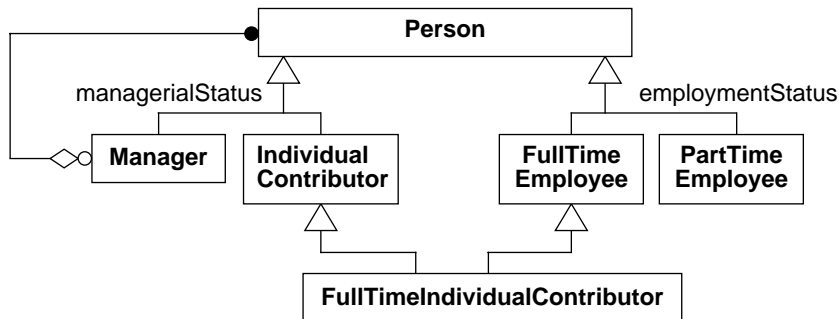


Figure 3.25 Multiple inheritance from different discriminators

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.

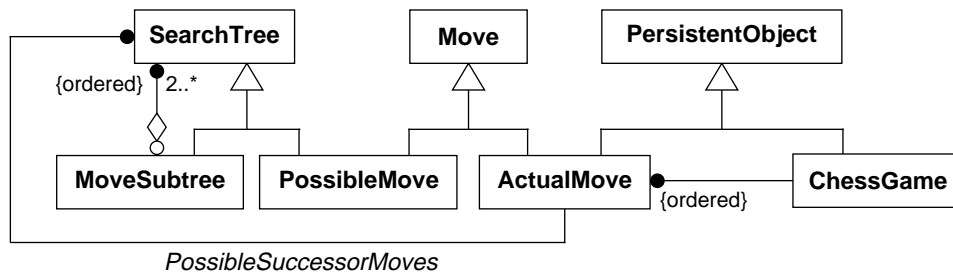


Figure 3.26 Multiple inheritance without a common ancestor

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.

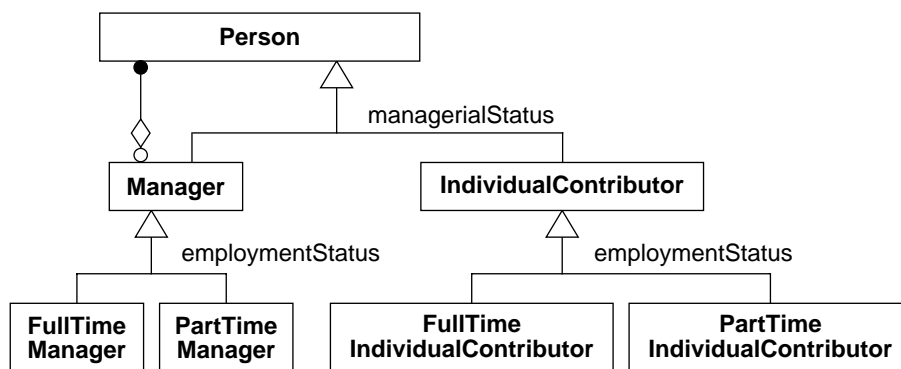


Figure 3.27 Multiple inheritance workaround: factoring

- **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.

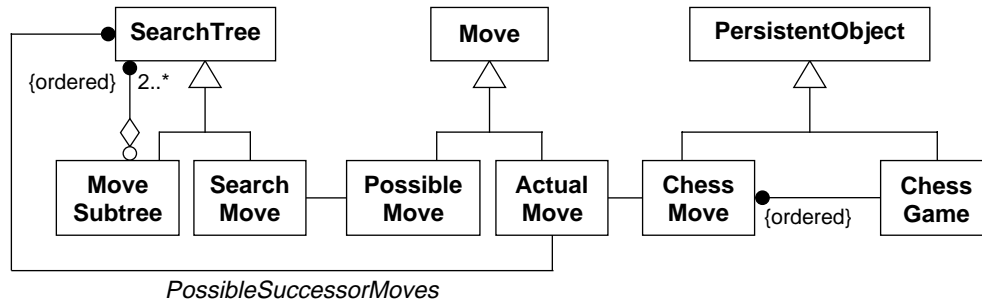


Figure 3.28 Multiple inheritance workaround: fragmenting a subclass

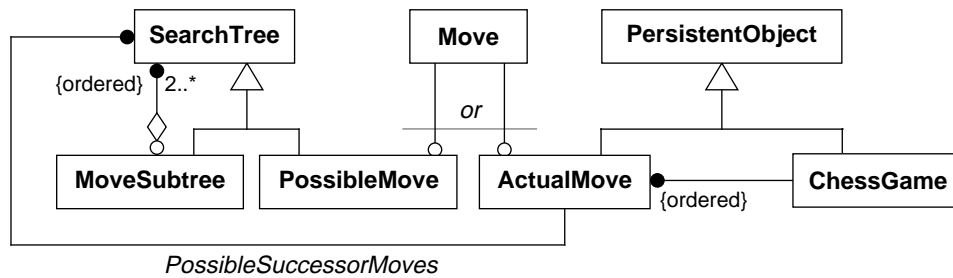


Figure 3.29 Multiple inheritance workaround: replacing generalization with exclusive-or associations

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

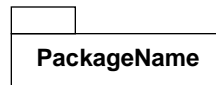


Figure 3.30 Notation for a package

have a *Customer* package or a *Part* package; *Customer* and *Part* are dominant classes that are important to the business of a corporation and appear in many applications. In an engineering application we used a dominant relationship, a large generalization for many kinds of equipment, to divide an object model into packages. Equipment was the focus of the model, and the attributes and relationships varied greatly across types of equipment. An object model of a compiler could be divided into packages for lexical analysis, parsing, semantic analysis, code generation, and optimization. Once some packages have been established, symmetry may suggest additional packages.

On the basis of our experience in creating packages, we can offer the following tips:

- **Carefully delineate each package’s scope.** The precise boundaries of a package are a matter of judgment. Like other aspects of modeling, defining the scope of a package requires planning and organization. Make sure that class and association names are unique within each package, and use consistent names across packages as much as possible.
- **Make packages cohesive.** There should be fewer associations between classes that appear in different packages than between classes that appear in a single package. Classes may appear in multiple packages, helping to bind them, but ordinarily associations and generalizations should appear in a single package.
- **Define each class in a single package.** The defining package should show the class name, attributes, and possibly operations. Other packages that refer to a class can use a class icon, a box that contains only the class name. This convention makes it easier to read object diagrams because a class is most prominent in its defining package. It ensures that readers of the object model will not become distracted by possibly inconsistent definitions or be misled by forgetting a prior class definition. This convention also makes it easier to develop packages concurrently.

3.6.1 Logical Horizon

You can often use a class with a large logical horizon as the nucleus for a package. The **logical horizon** [Feldman-86] of a class is the set of classes reachable by one or more paths terminating in a combined multiplicity of “one” or “zero or one.” A **path** is a sequence of consecutive associations and generalization levels. When computing the logical horizon, you may traverse a generalization hierarchy to obtain further information for a set of objects. You may not, however, traverse to sibling objects, such as by going up and then down the hierarchy. The logical horizons of various classes may, and often do, overlap. In Figure 2.23 on page 28 the logical horizon of *FlightDescription* is *Airport*, *Airline*, and *AircraftDescription*. The logical horizon of *Airport* is the empty set.

Figure 3.31 shows the computation of the logical horizon for *FlightReservation*.

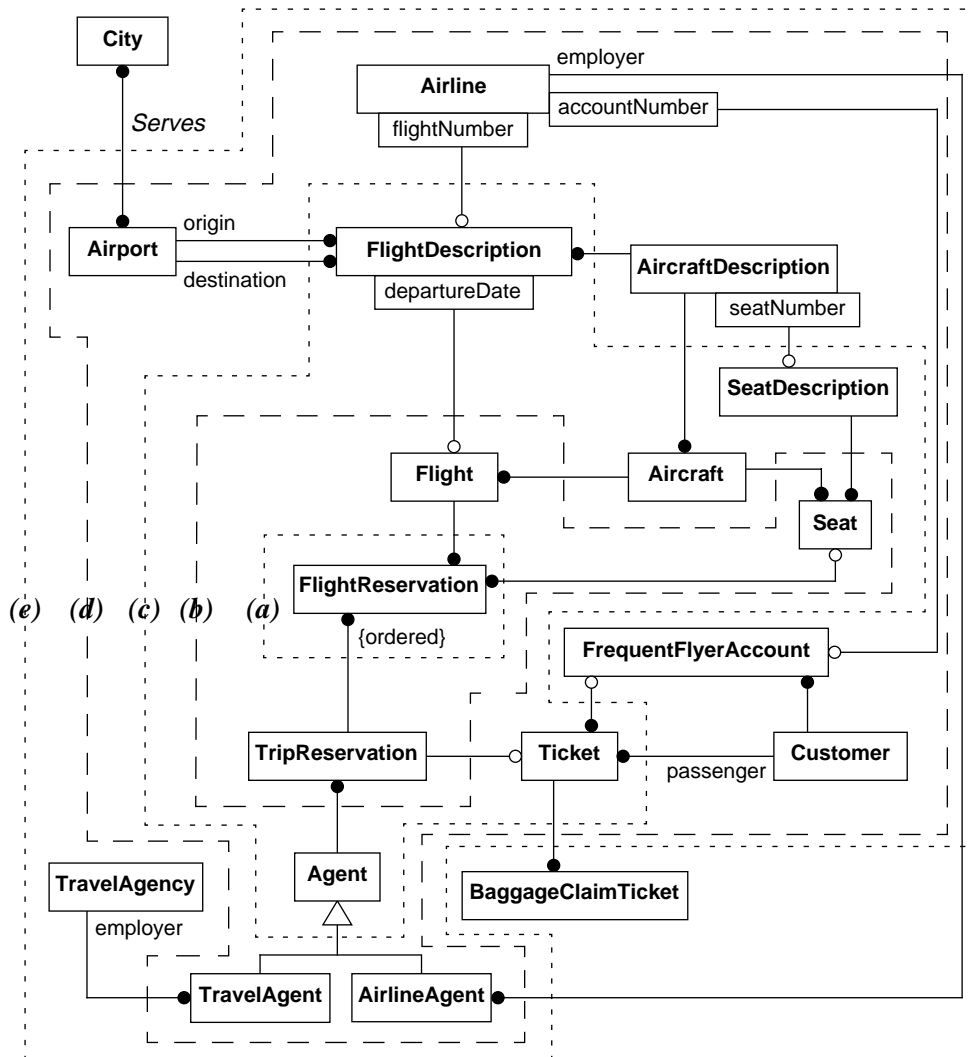


Figure 3.31 Computing the logical horizon for *FlightReservation*

- (a) We start with *FlightReservation*.
- (b) Each *FlightReservation* has a *Flight*, *Seat*, and *TripReservation*.
- (c) A *TripReservation* implies an *Agent* and a *Ticket*. A *Flight* implies a *FlightDescription* and an *Aircraft*. A *Seat* implies a *SeatDescription*.
- (d) A *Ticket* implies a *FrequentFlyerAccount* and a *Customer*; an *Agent* leads to *TravelAgent* and an *AirlineAgent* via generalization. The *FlightDescription* implies an *Airport*, *Airline*, and *AircraftDescription*.

- (e) A *TravelAgent* has a *TravelAgency* as an *employer*. Thus the logical horizon of *FlightReservation* includes every class in the diagram except *City* and *BaggageClaimTicket*.

When computing the logical horizon, you should disregard any qualifiers and treat the associations as if they were unqualified. The purpose of the logical horizon is to compute the objects that can be inferred from some starting object.

3.6.2 Example of Packages

Figure 3.32 shows a model for an airline information system with packages organized on a functional basis. (We are elaborating the model presented in Figure 2.23.) The reservations package records customer booking of airline travel. Flight operations deals with the actual logistics of planes arriving and departing. The aircraft information package stores seating layout and manufacturing data. Travel awards tracks bonus free travel for each customer; a person may submit a frequent flyer account number at the time of a reservation, but does not receive credits until after taking the flight. Baggage handling involves managing bags in conjunction with flights and accommodating errant pieces of luggage. Crew scheduling involves scheduling to staff flight needs. The subsequent diagrams elaborate all packages except *CrewScheduling*.

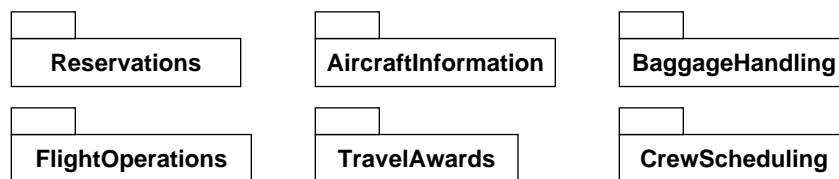
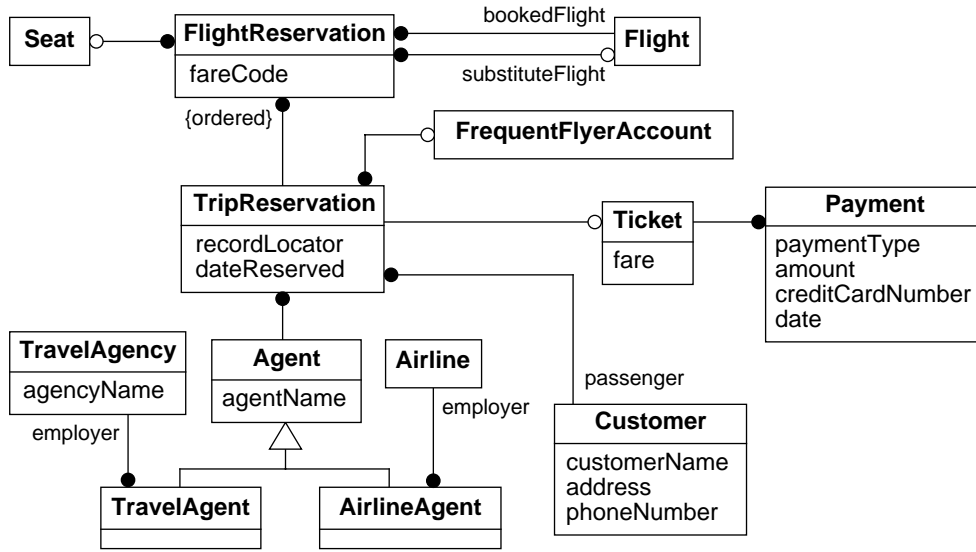


Figure 3.32 A partial high-level object model for an airline information system

Figure 3.33 describes the *Reservations* package. A trip reservation consists of a sequence of flight reservations, where each flight reservation refers to a specific flight. Sometimes another flight is substituted for a booked flight because of equipment problems, weather delays, or customer preference. The passenger may reserve a seat for each flight. A trip reservation is made on some date; the passenger must purchase a ticket within a certain number of days or the reservation becomes void. The airlines use record locators to find a particular trip reservation quickly and unambiguously. A trip is reserved by an agent, who either works for an airline or a travel agency. The frequent flyer account may be noted for a passenger. Although the structure of the model does not show it, the owner of the frequent flyer account must be the same as the passenger. We directly associate *TripReservation* with *FrequentFlyerAccount* and *Customer*, because a customer can make a reservation and specify a frequent flyer account before a ticket is even issued. Multiple payments may be made for a trip, such as two credit-card charges. Payment may also be made by cash or check.

Figure 3.34 describes the *FlightOperations* package. An airport serves many cities, and a city may have multiple airports. Airlines operate flights between airports. A flight descrip-



{The owner of the frequent flyer account must be the same as the passenger.}

Figure 3.33 An object model for the *Reservations* package

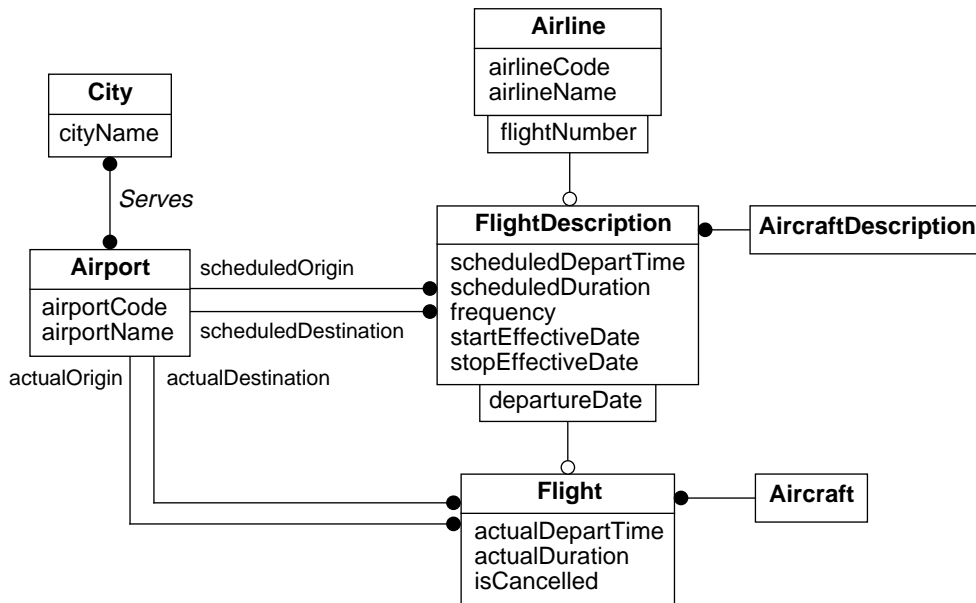


Figure 3.34 An object model for the *FlightOperations* package

tion 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 effectivity dates bracket the time period for which the published flight description is in effect. The actual origin, destination, departure time, and duration of a flight can vary because of weather and equipment problems.

Figure 3.35 presents a simple model of the *AircraftInformation* package. Each aircraft model has a manufacturer, model number, and specific numbering for seats. The seat type may be first class, business, or coach. Each individual aircraft has a registration number and refers to an aircraft model.

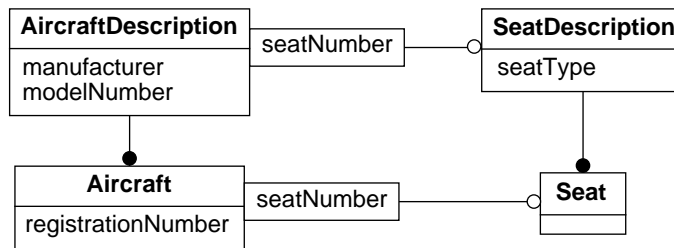


Figure 3.35 An object model for the *AircraftInformation* package

Figure 3.36 describes the *TravelAwards* package. A customer may have multiple frequent flyer accounts. Airlines identify each account with an account number. An account may receive numerous frequent flyer credits. Some frequent flyer credits pertain to flights; others (indicated by *creditType*) concern adjustments, redemption, long distance mileage, credit card mileage, hotel stays, car rental, and other kinds of inducements to patronize a business.

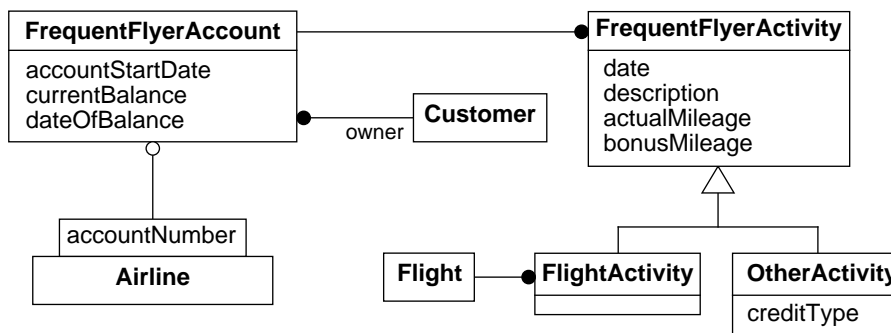


Figure 3.36 An object model for the *TravelAwards* package

Figure 3.37 describes the *BaggageHandling* package. A customer may check multiple bags for a trip and receives a claim ticket for each bag. Sometimes a bag is lost, damaged, or

delayed, in which case the customer completes a baggage complaint form for each problem bag.

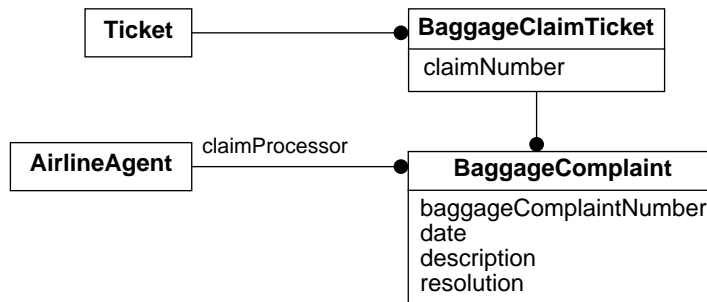


Figure 3.37 An object model for the *BaggageHandling* package

3.7 DERIVED DATA AND CONSTRAINTS

Derived data is data that can be completely determined from other data. Classes, attributes, and associations can all be derived. The underlying data can, in turn, be base data or further derived. Do not confuse our use of the term “derived” with the C++ derived class. A C++ derived class refers to the subclass of a generalization; it has nothing to do with OMT’s meaning of derived data.

As a rule, you should not show derived data during analysis unless the data appears in the problem description. During design you can add derived data to improve efficiency and ease implementation. During implementation you can compute derived data on demand from constituent data (lazy evaluation) or precompute and cache it (eager evaluation). Derived data that is precomputed must be marked as invalid or recomputed if constituent data is changed.

The notation for derived data is a slash preceding the name of the attribute, class, association, or role. Figure 3.38 shows an example of a derived attribute for airline flight descriptions. Exercise 3.8 illustrates derived associations.

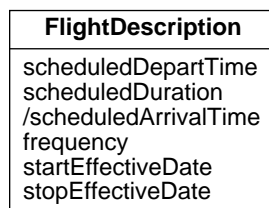


Figure 3.38 A derived attribute

A **constraint** is a functional relationship between modeling constructs such as classes, attributes, and associations. A constraint restricts the values of data. You may place simple constraints in the object model. You should specify complex constraints in the functional model.

A “good” model should capture many constraints with its very structure. In fact, the ability of a model to express important constraints is one measure of the quality of a model. (See Exercise 2.13.) Most object models require several iterations to strike a proper balance between rigor, simplicity, and elegance. However, sometimes it is not practical to express all important constraints with the structure of a model. For example, in Figure 3.33 we found it difficult to express structurally that the owner of the frequent flyer account must be the same as the passenger. In Figure 3.17 we specified that the catalog parts and physical parts aggregations must be acyclic.

Constraints are denoted by text in braces (“{” and “}”). The text of a constraint should clearly indicate the affected data. Similarly, comments are also delimited by braces. We often use comments to document the rationale for subtle modeling decisions and convey important enumerations.

Sometimes it is useful to draw a dotted arrow between classes or associations to indicate the scope of a constraint. For example, in Figure 3.39 a table has many columns; the primary key columns are a subset of the overall columns.

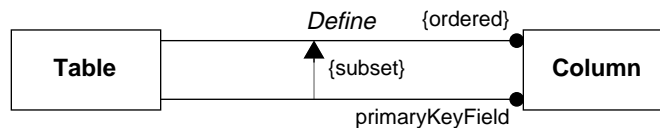


Figure 3.39 A subset constraint between associations

3.8 ADVANCED PRACTICAL TIPS

The following list summarizes the practical aspects of the object-modeling constructs described in this chapter.

- **Class attributes.** Try to avoid class attributes in your models. Often you can restructure a model, both avoiding class attributes and making the model more precise. (Section 3.1.2)
- **Domains.** Specify domains for attributes instead of data types. Domains promote uniform assignment of data types and convey additional semantic information. (Section 3.1.5)
- **Enumeration domains.** Do not create unnecessary generalizations for attributes of enumeration domain. Only specialize a class when the subclasses have distinct attributes, operations, or associations. (Section 3.1.5)

- ***N*-ary associations.** Try to avoid general ternary and *n*-ary associations. Most of these can be decomposed into binary associations, with possible qualifiers and link attributes. (Section 3.2.2)
- **Aggregation.** Consider aggregation when the “is-part-of” relationship holds. An aggregation must satisfy the transitivity and antisymmetry properties. Be careful not to confuse physical and catalog aggregation. (Section 3.3)
- **Roles.** Do not confuse classes with roles. A role is a use of a class in an association; a class may assume various roles. Do not introduce multiple classes in a model when there really is just one class with multiple roles. It is a good practice to label a class with its intrinsic name rather than a role name. (Section 3.4.2)
- **Multiple inheritance.** Try to avoid multiple inheritance during analysis because it is often confusing. Multiple inheritance is more helpful during design because of the need to mixin orthogonal aspects of objects. (Section 3.5)
- **Large models.** Organize large models so that the reader can understand portions of the model at a time, rather than the whole model at once. Packages are useful for organizing large models. (Section 3.6)
- **Constraints.** You may be able to restructure an object model to improve clarity and capture additional constraints. Use comments to express constraints that are awkward to represent with object-model structure. Also add comments to document modeling rationale and important enumeration values. (Section 3.7)

3.9 CHAPTER SUMMARY

Classes may have attributes and operations whose value is common to a group of objects. We advise that you restructure your object models to minimize use of class attributes. Attribute multiplicity specifies whether an attribute may be single or multivalued and whether an attribute is optional or mandatory. A domain is the set of possible values for an attribute. During design you should assign a domain to each attribute, instead of just directly assigning a data type.

The degree of an association is the number of distinct roles for each link. The vast majority of associations are binary or qualified binary. Ternary associations occasionally occur, but we have rarely encountered an association of higher degree.

A qualification cascade is a series of consecutive qualified associations. Qualification cascades often occur where an accumulation of qualifiers denotes increasingly specific objects.

Aggregation is a kind of association in which a whole, the assembly, is composed of parts, the components. 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 and antisymmetry properties. Do not confuse physical and catalog aggregation. With physical aggregation each component is dedicated to at most one assembly. With catalog aggregation components are reusable across multiple assemblies.

Generalization superclasses may or may not have direct instances. A concrete class can have direct instances; an abstract class has no direct instances. Multiple inheritance permits a class to inherit attributes, operations, and associations from multiple superclasses. Multiple inheritance brings greater modeling power but at the cost of greater conceptual and implementation complexity.

You will need multiple pages of diagrams to express object models for large problems. Large object models can be organized and made tractable with packages. Packages partition an object model into groups of tightly connected classes, associations, and generalizations.

Figure 3.40 lists the key concepts for this chapter.

abstract class	derived association	multiple inheritance
aggregation	derived attribute	package
association degree	derived class	path
attribute multiplicity	domain	physical aggregation
candidate key	enumeration domain	qualification cascade
catalog aggregation	exclusive-or association	secondary data
class attribute	instantiation	signature
class operation	logical horizon	structured domain
concrete class	maximum multiplicity	ternary association
constraint	minimum multiplicity	

Figure 3.40 Key concepts for Chapter 3

CHANGES TO OMT NOTATION

This chapter has introduced the following changes to the notation in [Rumbaugh-91] for compatibility with the UML notation [UML-98]. These changes are in addition to those from Chapter 2.

- **Instantiation.** 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 («»).
- **Class attribute and class operation.** You can indicate class attributes and class operations with an underline.
- **Exclusive-or association.** You can use a dashed line annotated by the legend “or” to group exclusive-or associations.
- **Compound qualifier.** You may use more than one attribute as the qualifier for an association.
- **Abstract and concrete classes.** You can indicate these by placing the legend *abstract* or *concrete* below the class name.
- **Package.** The notation for a package is a box with the addition of a tab.

- **Derived data.** Classes, attributes, and associations can all be derived. A slash in front of a name denotes derived data.

We have made some further minor notation extensions of our own.

- **Attribute multiplicity.** You can specify the number of values for an attribute within 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 collection with fixed limits $[lowerLimit..upperLimit]$.
- **Candidate key.** You can specify a candidate key for a class with the notation $\{CKn\}$ next to the participating attributes.
- **Structured domain.** You can use indentation to show the structure of domains.

BIBLIOGRAPHIC NOTES

Chapter 4 of [Booch-94] presents an insightful treatment of inheritance in the context of the broader classification literature.

This chapter contains much new material that complements our earlier book [Rumbaugh-91]. Our most significant improvements are in the areas of domains, secondary data, and aggregation.

REFERENCES

- [Blaha-93a] Michael Blaha. Secondary aspects of modeling. *Journal of Object-Oriented Programming* 6, 1 (March 1993), 15–18.
- [Blaha-93b] Michael Blaha. Aggregation of Parts of Parts of Parts. *Journal of Object-Oriented Programming* 6, 5 (September 1993), 14–20.
- [Booch-94] Grady Booch. *Object-Oriented Analysis and Design with Applications*. Reading, Massachusetts: Benjamin/Cummings, 1994.
- [Cattell-96] RGG Cattell, editor. *The Object Database Standard: ODMG-93, Release 1.2*. San Francisco, California: Morgan-Kaufmann, 1996.
- [Feldman-86] P Feldman and D Miller. Entity model clustering: Structuring a data model by abstraction. *Computer Journal* 29, 4 (August 1986), 348–360.
- [Rumbaugh-88] James Rumbaugh. Controlling propagation of operations using attributes on relations. *OOPSLA'88 as ACM SIGPLAN* 23, 11 (November 1988), 285–296.
- [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.
- [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.

EXERCISES

- 3.1** (2) Add domains to the object model in Figure E2.1.
- 3.2** (3) Figure 3.16 is an example of catalog aggregation. Construct an instance diagram for Figure 3.17 using the instances in Figure 3.16. Assume that the *role* shown in Figure 3.17 is *normal*.
- 3.3** (3) The object model in Figure E3.1 describes a reporting hierarchy within a company. Change the object model to accommodate matrix management (where a person may report to more than one manager).

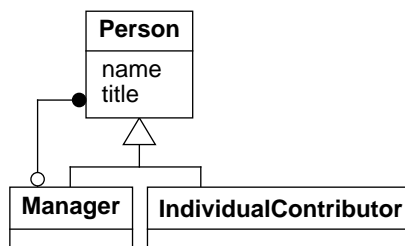


Figure E3.1 An object model for the management hierarchy in a corporation

- 3.4** (7) Extend the object model in Figure E3.1 to track the evolution of the reporting hierarchy over time. The reporting hierarchy changes as persons join and leave a company. The hierarchy also changes due to promotions and demotions.
- 3.5** (4) What is the logical horizon of *Statement* in Figure E2.1?
- 3.6** (4) What is the logical horizon of *City* in Figure 3.34?
- 3.7** (5) What is the logical horizon of *OtherActivity* in Figure 3.36? You should also consider relationships for the other packages in Section 3.6.2.
- 3.8** (7) Add the following derived information to extend your answer to Exercise 2.16: age, grandparent, ancestor, descendant, aunt, uncle, sibling, and cousin.
- 3.9** (9) Construct an object model that describes the baseball statistics listed in Figure E3.2. Use any baseball knowledge you may have. Even if you are unfamiliar with the game of baseball, this exercise is still useful. For legacy applications, it is not uncommon to be given examples of data structure with little explanation. We have chosen data that illustrates most multiplicity combinations that would be found in a more comprehensive set of data. Note that the St. Louis Browns moved at the end of the 1953 season and became the Baltimore Orioles; the Philadelphia Athletics moved at the end of the 1954 season and became the Kansas City Athletics.
- 3.10** (9) Prepare an object model for the game of hearts. A game of hearts typically involves four players. The objective of the game is to play a series of hands and score the fewest points. There are two phases to each hand: exchanging cards and then playing cards.
 The following cycle must be observed for exchanging cards. For the first hand each player passes three cards to the player on the left (and receives three cards from the player on the right). For the second hand each player passes three cards to the right. Each player passes three cards

Player Statistics—Batting

Year	League	City	Team name	Player	Field position	Bat pos.	At bat	HR	RBI	BA
1953	American	St. Louis	Browns	Vern Stephens	3B	R	165	4	17	.321
1953	American	Chicago	White Sox	Vern Stephens	3B,SS	R	129	1	14	.186
1953	American	St. Louis	Browns	Bob Elliott	3B	R	160	5	29	.250
1953	American	Chicago	White Sox	Bob Elliott	3B,OF	R	208	4	32	.260
1953	American	Phil.	Athletics	Dave Philley	OF,3B	B	620	9	59	.303
1953	American	St. Louis	Browns	Don Larsen	P,OF	R	81	3	10	.284
1955	American	Detroit	Tigers	Al Kaline	OF	R	588	27	102	.340
1955	American	Detroit	Tigers	Fred Hatfield	2B,3B,SS	L	413	8	33	.232

Player Statistics—Pitching

Year	League	City	Team name	Player	Pitch pos.	IP	Win	Loss	Save	ERA
1953	American	St. Louis	Browns	Don Larsen	R	193	7	12	2	4.15
1953	American	St. Louis	Browns	Bobo Holloman	R	65	3	7	0	5.26
1953	American	St. Louis	Browns	Satchel Paige	R	117	3	9	11	3.54
1953	American	Phil.	Athletics	Alex Kellner	L	202	11	12	0	3.92

Team Statistics

Year	League	City	Team name	Win	Loss	Manager	Save	ERA	AB	HR	RBI	BA
1953	Amer.	St. L	Browns	54	100	Marty Marion	24	4.48	5264	112	522	.249
1953	Amer.	Phil.	Athletics	59	95	Jimmy Dykes	11	4.67	5455	116	588	.256
1953	Amer.	Det.	Tigers	60	94	Fred Hutchinson	16	5.25	5553	108	660	.266
1955	Amer.	Balt.	Orioles	57	97	Paul Richards	22	4.21	5257	54	503	.240
1955	Amer.	KC	Athletics	63	91	Lou Boudreau	23	5.35	5335	121	593	.261
1955	Amer.	Det.	Tigers	79	75	Bucky Harris	12	3.79	5283	130	724	.266
1953	Natl.	Phil.	Phillies	83	71	Steve O'Neill	15	3.80	5290	115	657	.265
1953	Natl.	St. L	Cardinals	83	71	Eddie Stanky	36	4.23	5397	140	722	.273
1953	Natl.	NY	Giants	70	84	Leo Durocher	20	4.25	5362	176	739	.271
1955	Natl.	NY	Giants	80	74	Leo Durocher	14	3.77	5288	169	643	.260
1955	Natl.	Phil.	Phillies	77	77	Mayo Smith	21	3.93	5092	132	631	.255
1955	Natl.	St. L	Cardinals	17	19	Eddie Stanky	15	4.56	5266	143	608	.261
1955	Natl.	St. L	Cardinals	51	67	Harry Walker						

Figure E3.2 Sample baseball statistics

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.