3

Object Modeling

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

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

3.1 OBJECTS AND CLASSES

3.1.1 Objects

The purpose of object modeling is to describe objects. For example, *Joe Smith*, *Simplex company*, *Eassie*, *process number 7648*, and *the top window* are objects. An object is simply something that makes sense in an application context.

We define an *object* as a concept, abstraction, or thing with crisp boundaries and meaning for the problem at hand. Objects serve two purposes: They promote understanding of the real world and provide a practical basis for computer implementation. Decomposition of a problem into objects depends on judgment and the nature of the problem. There is no one correct representation.

All objects have identity and are distinguishable. Two apples with the same color, shape, and texture are still individual apples; a person can eat one and then eat the other. Similarly, identical twins are two distinct persons, even though they may look the same. The term *identity* means that objects are distinguished by their inherent existence and not by descriptive properties that they may have.

The word *object* is often vaguely used in the literature. Sometimes *object* means a single thing, other times it refers to a group of similar things. Usually the context resolves any ambiguity. When we want to be precise and refer to exactly one thing, we will use the phrase *object instance*. We will use the phrase *object class* to refer to a group of similar things.

3.1.2 Classes

An *object class* describes a group of objects with similar properties (attributes), common behavior (operations), common relationships to other objects, and common semantics. *Person*, *company*, *animal*, *process*, and *window* are all object classes. Each person has an age, IQ, and may work at a job. Each process has an owner, priority, and list of required resources. Objects and object classes often appear as nouns in problem descriptions.

The abbreviation *class* is often used instead of *object class*. Objects in a class have the same attributes and behavior patterns. Most objects derive their individuality from differences in their attribute values and relationships to other objects. However, objects with identical attribute values and relationships are possible.

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

Each object "knows" its class. Most object-oriented programming languages can determine an object's class at run time. An object's class is an implicit property of the object.

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

3.1.3 Object Diagrams

We began this chapter by discussing some basic modeling concepts, specifically *object* and *class*. We have described these concepts with examples and prose. Since this approach is

vague for more complex topics, we need a formalism for expressing object models that is coherent, precise, and easy to formulate.

Object diagrams provide a formal graphic notation for modeling objects, classes, and their relationships to one another. Object diagrams are useful both for abstract modeling and for designing actual programs. Object diagrams are concise, easy to understand, and work well in practice. We use object diagrams throughout this book. New concepts are illustrated by object diagrams to introduce the notation and clarify our explanation of concepts. There are two types of object diagrams: class diagrams and instance diagrams.

A *class diagram* is a schema, pattern, or template for describing many possible instances of data. A class diagram describes object classes.

An *instance diagram* describes how a particular set of objects relate to each other. An instance diagram describes object instances. Instance diagrams are useful for documenting test cases (especially scenarios) and discussing examples. A given class diagram corresponds to an infinite set of instance diagrams.

Figure 3.1 shows a class diagram (left) and one possible instance diagram (right) described by it. Objects *Joe Smith*, *Mary Sharp*, and an anonymous person are instances of class *Person*. The OMT symbol for an object instance is a rounded box. The class name in parentheses is at the top of the object box in boldface. Object names are listed in normal font. The OMT symbol for a class is a box with class name in boldface.

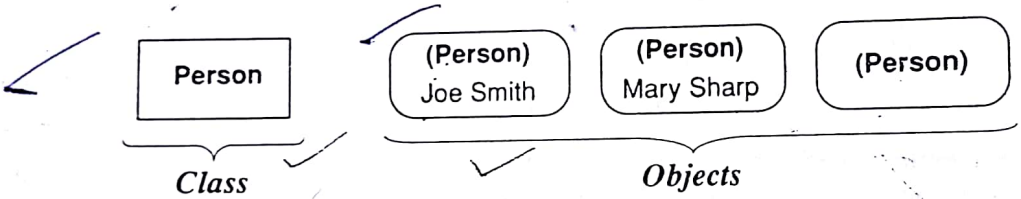


Figure 3.1 Class and objects

Class diagrams describe the general case in modeling a system. Instance diagrams are used mainly to show examples to help to clarify a complex class diagram. The distinction between class diagrams and instance diagrams is in fact artificial; classes and instances can appear on the same object diagram, but in general it is not useful to mix classes and instances. (The exception is metadata, discussed in Section 4.5.)

3.1.4 Attributes

An *attribute* is a data value held by the objects in a class. *Name*, *age*, and *weight* are attributes of *Person* objects. *Color*, *weight*, and *model-year* are attributes of *Car* objects. Each attribute has a value for each object instance. For example, attribute *age* has value "24" in object *Joe Smith*. Paraphrasing, Joe Smith is 24 years old. Different object instances may have the same or different values for a given attribute. Each attribute name is unique within a class (as opposed to being unique across all classes). Thus class *Person* and class *Company* may each have an attribute called *address*.

An attribute should be a pure data value, not an object. Unlike objects, pure data values do not have identity. For example, all occurrences of the integer "17" are indistinguishable,

as are all occurrences of the string "Canada." The country Canada is an object, whose *name* attribute has the value "Canada" (the string). The capital of Canada is a city object and should not be modeled as an attribute, but rather as an association between a country object and a city object (explained in Section 3.2). The *name* of this city object is "Ottawa" (the string).

Attributes are listed in the second part of the class box. Each attribute name may be followed by optional details, such as type and default value. The type is preceded by a colon. The default value is preceded by an equal sign. At times, you may choose to omit showing attributes in class boxes. It depends on the level of detail desired in the object model. Class boxes have a line drawn between the class name and attributes. Object boxes do not have this line in order to further differentiate them from class boxes.

Figure 3.2 shows object modeling notation. Class *Person* has attributes *name* and *age*. *Name* is a string and *age* is an integer. One object in class *Person* has the value *Joe Smith* for name and the value *24* for age. Another object has name *Mary Sharp* and age *52*.

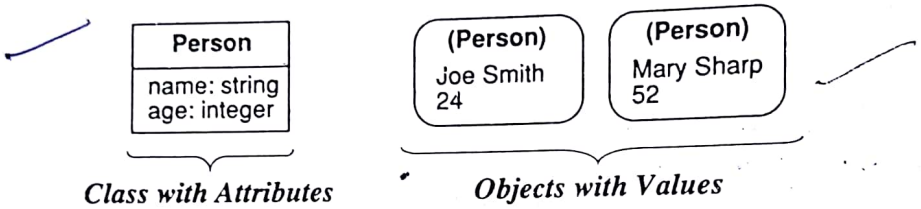


Figure 3.2 Attributes and values

Some implementation media, such as many databases, require an object to have a unique identifier that identifies each object. Explicit object identifiers are not required in an object model. Each object has its own unique identity. Most object-oriented languages automatically generate implicit identifiers with which to reference objects. You need not and should not explicitly list identifiers. Figure 3.3 emphasizes this point. Identifiers are a computer artifact and have no intrinsic meaning beyond identifying an object.

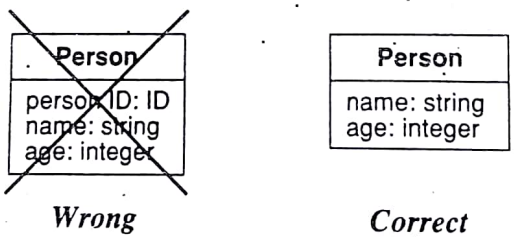


Figure 3.3 Do not explicitly list object identifiers

Do not confuse internal identifiers with real-world attributes. Internal identifiers are purely an implementation convenience and have no meaning in the problem domain. For example, social security number, license plate number, and telephone number are not internal

identifiers because they have meaning in the real world. Social security number, license plate number, and telephone number are legitimate attributes.

3.1.5 Operations and Methods

An *operation* is a function or transformation that may be applied to or by objects in a class. *Hire*, *fire*, and *pay-dividend* are operations on class *Company*. *Open*, *close*, *hide*, and *redisplay* are operations on class *Window*. All objects in a class share the same operations.

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

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

An operation may have arguments in addition to its target object. Such arguments parameterize the operation but do not affect the choice of method. The method depends only on the class of the target object. (A few object-oriented languages, notably CLOS, permit the choice of method to depend on any number of arguments, but such generality leads to considerable semantic complexity, which we shall not explore.)

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

Operations are listed in the lower third of the class box. Each operation name may be followed by optional details, such as argument list and result type. An argument list is written in parentheses following the name; the arguments are separated by commas. The name and type of each argument may be given. The result type is preceded by a colon and should not be omitted, because it is important to distinguish operations that return values from those that do not. An empty argument list in parentheses shows explicitly that there are no arguments; otherwise no conclusions can be drawn. Operations may be omitted from high-level diagrams.

In Figure 3.4, the class *Person* has attributes *name* and *age* and operations *change-job* and *change-address*. *Name*, *age*, *change-job*, and *change-address* are features of *Person*.

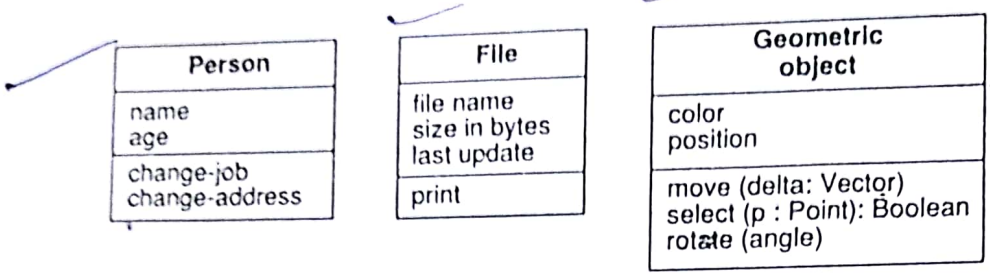


Figure 3.4 Operations

Feature is a generic word for either an attribute or operation. Similarly, *File* has a *print* operation. *Geometric object* has *move*, *select*, and *rotate* operations. *Move* has argument *delta*, which is a *Vector*; *select* has one argument *p* which is of type *Point* and returns a *Boolean*; and *rotate* has argument *angle*.

During modeling, it is useful to distinguish operations that have side effects from those that merely compute a functional value without modifying any objects. The latter kind of operation is called a *query*. Queries with no arguments except the target object may be regarded as derived attributes. For example, the width of a box can be computed from the positions of its sides. A derived attribute is like an attribute in that it is a property of the object itself, and computing it does not change the state of the object. In many cases, an object has a set of attributes whose values are interrelated, of which only a fixed number of values can be chosen independently. An object model should generally distinguish independent *base attributes* from dependent *derived attributes*. The choice of base attributes is arbitrary but should be made to avoid overspecifying the state of the object. The remaining attributes may be omitted or may be shown as derived attributes as described in Section 4.7.4.

3.1.6 Summary of Notation for Object Classes

Figure 3.5 summarizes object modeling notation for classes. A class is represented by a box which may have as many as three regions. The regions contain, from top to bottom: class name, list of attributes, and list of operations. Each attribute name may be followed by optional details such as type and default value. Each operation name may be followed by optional details such as argument list and result type. Attributes and operations may or may not be shown; it depends on the level of detail desired.

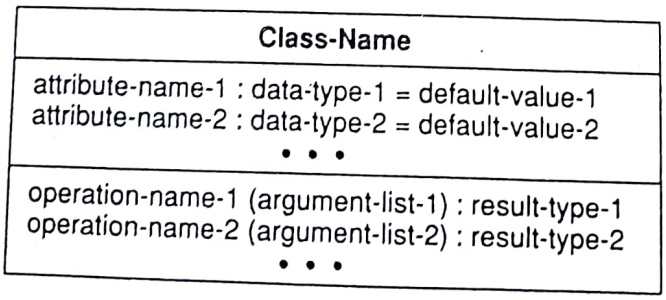


Figure 3.5 Summary of object modeling notation for classes

3.2 LINKS AND ASSOCIATIONS

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

3.2.1 General Concepts

A *link* is a physical or conceptual connection between object instances. For example, Joe Smith *Works-for* Simplex company. Mathematically, a link is defined as a tuple, that is, an ordered list of object instances. A link is an instance of an association.

An *association* describes a group of links with common structure and common semantics. For example, a person *Works-for* a company. All the links in an association connect objects from the same classes. Associations and links often appear as verbs in a problem statement. An association describes a set of potential links in the same way that a class describes a set of potential objects.

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

Associations are often implemented in programming languages as pointers from one object to another. A *pointer* is an attribute in one object that contains an explicit reference to another object. For example, a data structure for *Person* might contain an attribute *employer* that points to a *Company* object, and a *Company* object might contain an attribute *employees* that points to a set of *Employee* objects. Implementing associations as pointers is perfectly acceptable, but associations should not be modeled this way.

A link shows a relationship between two (or more) objects. Modeling a link as a pointer disguises the fact that the link is not part of either object by itself, but depends on both of them together. A company is not part of a person, and a person is not part of a company. Furthermore, using a pair of matched pointers, such as the pointer from *Person* to *Company* and the pointer from *Company* to a set of *Employee*, hides the fact that the forward and inverse pointers are dependent on each other. All connections among classes should therefore be modeled as associations, even in designs for programs. We must stress that associations are not just database constructs, although relational databases are built on the concept of associations.

Although associations are modeled as bidirectional they do not have to be implemented in both directions. Associations can easily be implemented as pointers if they are only traversed in a single direction. Chapter 10 discusses some trade-offs to consider when implementing associations.

Figure 3.6 shows a one-to-one association and corresponding links. Each association in the class diagram corresponds to a set of links in the instance diagram, just as each class corresponds to a set of objects. Each country has a capital city. *Has-capital* is the name of the

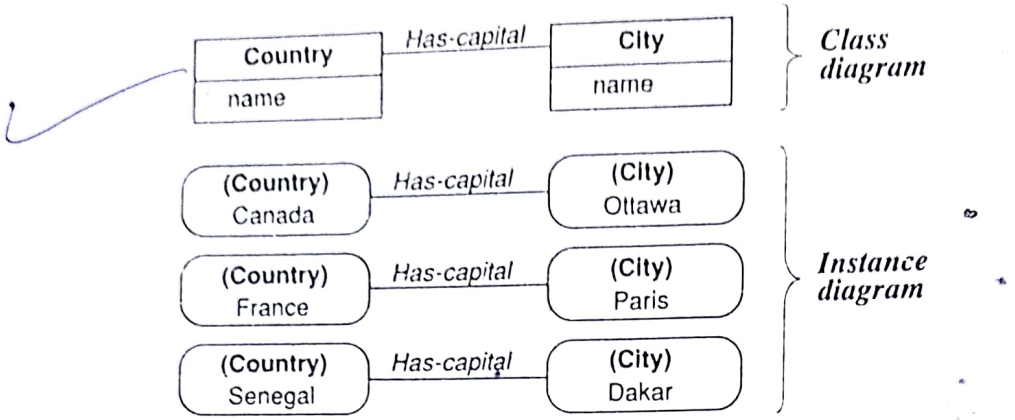


Figure 3.6 One-to-one association and links

association. The OMT notation for an association is a line between classes. A link is drawn as a line between objects. Association names are italicized. An association name may be omitted if a pair of classes has a single association whose meaning is obvious. It is good to arrange the classes to read from left-to-right, if possible.

Figure 3.7 is a fragment of an object model for a program. A common task that arises in computer-aided design (CAD) applications is to find connectivity networks: Given a line, find all intersecting lines; given an intersection point, find all lines that pass through it; given an area on the screen, find all intersection points. (We use the word *line* here to mean a finite line segment.)

In the class diagram, each point denotes the intersection of two or more lines; each line has zero or more intersection points. The instance diagram shows one possible set of lines. Lines *L1*, *L2*, and *L3* intersect at point *P1*. Lines *L3* and *L4* intersect at point *P2*. Line *L5* has no intersection points and thus has no link. The solid balls and "2+" are multiplicity symbols. Multiplicity specifies how many instances of one class may relate to each instance of another class and is discussed in the next section.

Associations may be binary, ternary, or higher order. In practice, the vast majority are binary or qualified (a special form of ternary discussed later). We have encountered a few general ternary and few, if any, of order four or more. Higher order associations are more complicated to draw, implement, and think about than binary associations and should be avoided if possible.

Figure 3.8 shows a *ternary* association: Persons who are programmers use computer languages on projects. This ternary association is an atomic unit and cannot be subdivided into binary associations without losing information. A programmer may know a language and work on a project, but might not use the language on the project. The OMT symbol for general ternary and n-ary associations is a diamond with lines connecting to related classes. The name of the association is written next to the diamond. Note that we did not name the association or links in Figure 3.8. Association names are optional and a matter of modeling judgment. Associations are often left unnamed when they can be easily identified by their classes. (This convention does not work if there are multiple associations between the same classes.)

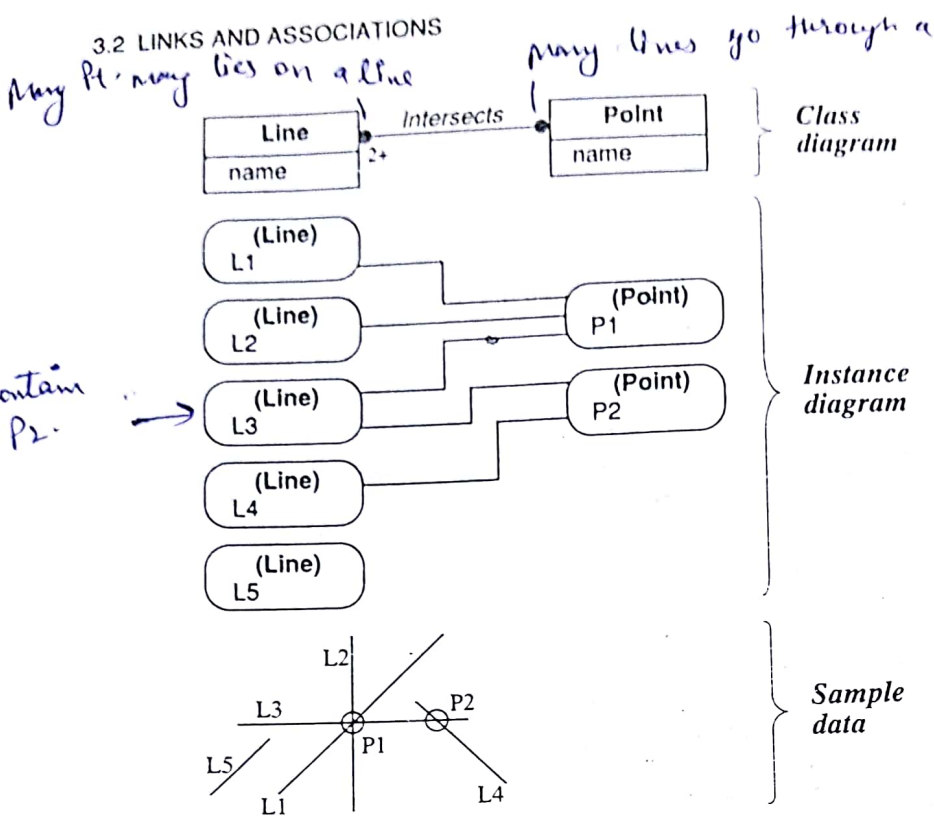


Figure 3.7 Many-to-many association and links

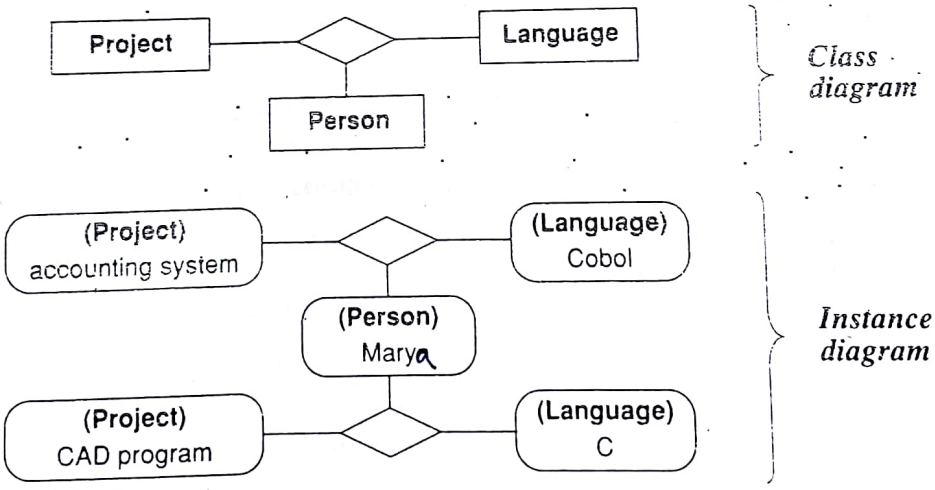


Figure 3.8 Ternary association and links

3.2.2 Multiplicity

Multiplicity specifies how many instances of one class may relate to a single instance of an associated class. Multiplicity constrains the number of related objects. Multiplicity is often described as being "one" or "many," but more generally it is a (possibly infinite) subset of the non-negative integers. Generally the multiplicity value is a single interval, but it may be a set of disconnected intervals. For example, the number of doors on a sedan is 2 or 4. Object diagrams indicate multiplicity with special symbols at the ends of association lines. In the most general case, multiplicity can be specified with a number or set of intervals, such as "1" (exactly one), "1+" (one or more), "3-5" (three to five, inclusive), and "2,4,18" (two, four, or eighteen). There are special line terminators to indicate certain common multiplicity values. A solid ball is the OMT symbol for "many," meaning zero or more. A hollow ball indicates "optional," meaning zero or one. A line without multiplicity symbols indicates a one-to-one association. In the general case, the multiplicity is written next to the end of the line, for example, "1+" to indicate one or more.

Reviewing our past examples, Figure 3.6 illustrates one-to-one multiplicity. Each country has one capital city. A capital city administers one country. (In fact, some countries, such as Netherlands and Switzerland, have more than one capital city for different purposes. If this fact were important, the model could be modified by changing the multiplicity or by providing a separate association for each kind of capital city.)

The association in Figure 3.7 exhibits many-to-many multiplicity. A line may have zero or more intersection points. An intersection point may be associated with two or more lines. In this particular case, *L1*, *L2*, and *L4* have one intersection point; *L3* has two intersection points; *L5* has no intersection points. *P1* intersects with three lines; *P2* intersects with two lines.

Figure 3.9 illustrates zero-or-one, or optional, multiplicity. A workstation may have one of its windows designated as the console to receive general error messages. It is possible, however, that no console window exists. (The word "console" on the diagram is a role name, discussed in Section 3.3.3.)

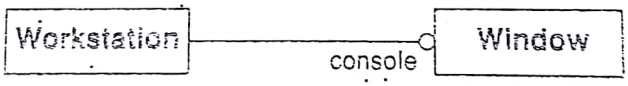


Figure 3.9 Zero-or-one multiplicity

Multiplicity depends on assumptions and how you define the boundaries of a problem. Vague requirements often make multiplicity uncertain. You should not worry excessively about multiplicity early in software development. First determine objects, classes, and associations, then decide on multiplicity.

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

The most important multiplicity distinction is between "one" and "many." Underestimating multiplicity can restrict the flexibility of an application. For example, many phone number utility programs are unable to accommodate persons with multiple phone numbers. On the other hand, overestimating multiplicity imposes extra overhead and requires the application to supply additional information to distinguish among the members of a "many" set. In a true hierarchical organization, for example, it is better to represent "boss" with a multiplicity of "zero or one," rather than allow for nonexistent matrix management.

This chapter only considers multiplicity for binary associations. The solid and hollow ball notation is ambiguous for n-ary ($n > 2$) associations, for which multiplicity is a more complex topic. Section 4.6 extends our treatment of multiplicity to n-ary associations.

3.2.3 The Importance of Associations

The notion of an association is certainly not a new concept. Associations have been widely used throughout the database modeling community for years. (See Chapter 12 for details.) In contrast, few programming languages explicitly support associations. We nevertheless emphasize that associations are a useful modeling construct for programs as well as databases and real-world systems, regardless of how they are implemented. During conceptual modeling, you should not bury pointers or other object references inside objects as attributes. Instead you should model them as associations to indicate that the information they contain is not subordinate to a single class, but depends on two or more classes [Rumbaugh-87].

Some object-oriented authors feel that every piece of information should be attached to a single class, and they argue that associations violate encapsulation of information into classes. We do not agree with this viewpoint. Some information inherently transcends a single class, and the failure to treat associations on an equal footing with classes can lead to programs containing hidden assumptions and dependencies.

Most object-oriented languages implement associations with object pointers. Pointers can be regarded as an implementation optimization introduced during the later stages of design. It is also possible to implement association objects directly, but the use of association objects during implementation is really a design decision (see Chapter 10).

3.3 ADVANCED LINK AND ASSOCIATION CONCEPTS

3.3.1 Link Attributes

An attribute is a property of the objects in a class. Similarly, a *link attribute* is a property of the links in an association. In Figure 3.10, *access permission* is an attribute of *Accessible by*. Each link attribute has a value for each link, as illustrated by the sample data at the bottom

* The term *association* as used in this book is synonymous with the term *relation* used in [Rumbaugh-87] and with the use of the term *relation* as used in discrete mathematics. We have used the term *association* to avoid confusion with the more restricted use of the term *relation* as used in relational databases, which usually permit relations between pure values only, not between objects with identity.

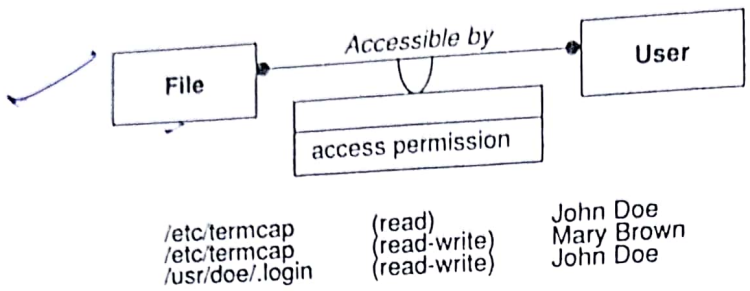


Figure 3.10 Link attribute for a many-to-many association

of the figure. The OMT notation for a link attribute is a box attached to the association by a loop; one or more link attributes may appear in the second region of the box. This notation emphasizes the similarity between attributes for objects and attributes for links.

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 attached to either object. In Figure 3.10, *access permission* is a joint property of *File* and *User*, and cannot be attached to either *File* or *User* alone without losing information.

Figure 3.11 presents link attributes for two many-to-one associations. Each person working for a company receives a salary and has a job title. The boss evaluates the performance of each worker. Link attributes may also occur for one-to-one associations.

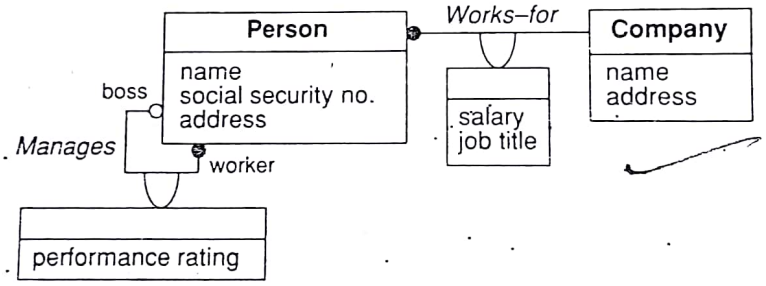
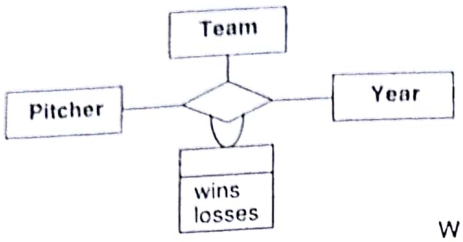


Figure 3.11 Link attributes for one-to-many associations

Figure 3.12 shows link attributes for a ternary association. A pitcher may play for many teams in a given year. A pitcher may also play many years for the same team. Each team has many pitchers. For each combination of team and year, a pitcher has a won-loss record. Thus for instance, Harry Eisenstat pitched for the Cleveland Indians in 1939, winning 6 games and losing 7 games.

Figure 3.13 shows how it is possible to fold link attributes for one-to-one and one-to-many associations into the class opposite the "one" side. This is not possible for many-to-many associations. As a rule, link attributes should not be folded into a class because future flexibility is reduced if the multiplicity of the association should change. Either form in Figure 3.13 can express a one-to-many association. However, only the link attribute form remains correct if the multiplicity of *Works-for* is changed to many-to-many.



			W	L
Harry Eisenstat	Cleveland Indians	1939	6	7
Harry Eisenstat	Detroit Tigers	1939	2	2
Willis Hudlin	Cleveland Indians	1939	9	10
Willis Hudlin	Cleveland Indians	1940	2	1
Willis Hudlin	Washington Senators	1940	1	2
Willis Hudlin	St. Louis Browns	1940	0	1

Figure 3.12 Link attributes for a ternary association

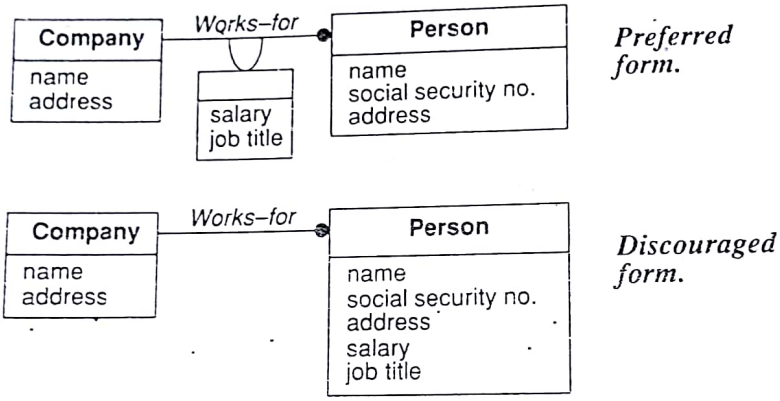


Figure 3.13 Link attribute versus object attribute

3.3.2 Modeling an Association as a Class

Sometimes it is useful to model an association as a class. Each link becomes one instance of the class. The link attribute box introduced in the previous section is actually a special case of an association as a class, and may have a name and operations in addition to attributes. Figure 3.14 shows the authorization information for users on workstations. Users may be authorized on many workstations. Each authorization carries a priority and access privileges, shown as link attributes. A user has a home directory for each authorized workstation, but the same home directory can be shared among several workstations or among several users. The home directory is shown as a many-to-one association between the authorization class and the directory class. It is useful to model an association as a class when links can participate in associations with other objects or when links are subject to operations.

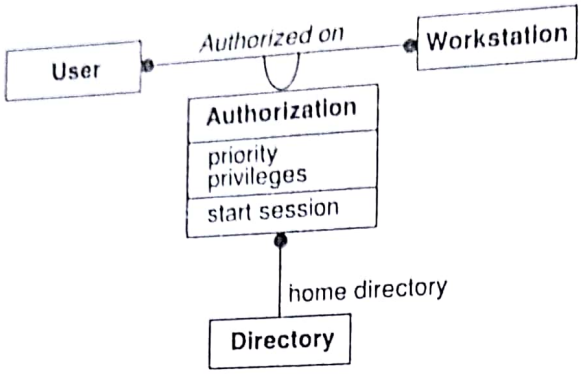
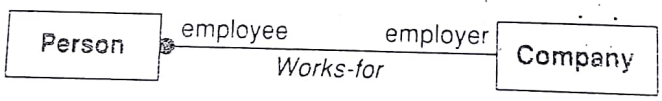


Figure 3.14 Modeling an association as a class

3.3.3 Role Names

A *role* is one end of an association. A binary association has two roles, each of which may have a *role name*. A *role name* is a name that uniquely identifies one end of an association. Roles provide a way of viewing a binary association as a traversal from one object to a set of associated objects. Each role on a binary association identifies an object or set of objects associated with an object at the other end. From the point of view of the object, traversing the association is an operation that yields related objects. The role name is a derived attribute whose value is a set of related objects. Use of role names provides a way of traversing associations from an object at one end, without explicitly mentioning the association. Roles often appear as nouns in problem descriptions.

Figure 3.15 specifies how *Person* and *Company* participate in association *Works-for*. A person assumes the role of *employee* with respect to a company; a company assumes the role of *employer* with respect to a person. A role name is written next to the association line near the class that plays the role (that is, the role name appears on the destination end of the traversal). Use of role names is optional, but it is often easier and less confusing to assign role names instead of, or in addition to, association names.



employee	employer
Joe Doe	Simplex
Mary Brown	Simplex
Jean Smith	United Widgets

Figure 3.15 Role names for an association

Role names are necessary for associations between two objects of the same class. For example, *boss* and *worker* distinguish the two employees participating in the *Manages* association in Figure 3.11. Role names are also useful to distinguish between two associations between the same pair of classes. When there is only a single association between a pair of

distinct classes, the names of the classes often serve as good role names, in which case the role names may be omitted on the diagram.

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

Figure 3.16 shows both uses of role names. A directory may contain many other directories and may optionally be contained in another directory. Each directory has exactly one user who is an owner, and many users who are authorized to use the directory.

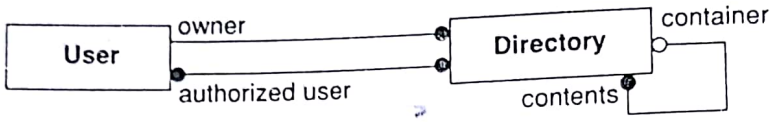


Figure 3.16 Role names for a directory hierarchy

An n-ary association has a role for each end. The role names distinguish the ends of the association and are necessary if a class participates in an n-ary association more than once. Associations of degree 3 or more cannot simply be traversed from one end to another as binary associations can, so the role names do not represent derived attributes of the participating classes. For example, in Figure 3.12, both a team and a year are necessary to obtain a set of pitchers.

3.3.4 Ordering

Usually the objects on the "many" side of an association have no explicit order, and can be regarded as a set. Sometimes, however, the objects are explicitly ordered. For example, Figure 3.17 shows a workstation screen containing a number of overlapping windows. The windows are explicitly ordered, so only the topmost window is visible at any point on the screen. The ordering is an inherent part of the association. An ordered set of objects on the "many" end of an association is indicated by writing "{ordered}" next to the multiplicity dot for the role. This is a special kind of constraint. (See Section 4.7 for a discussion of constraints.)

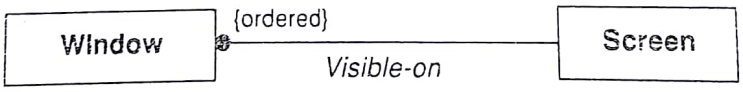


Figure 3.17 Ordered sets in an association

3.3.5 Qualification

A *qualified association* relates two object classes and a *qualifier*. The qualifier is a special attribute that reduces the effective multiplicity of an association. One-to-many and many-to-many associations may be qualified. The qualifier distinguishes among the set of objects at

the many end of an association. A qualified association can also be considered a form of ternary association.

For example, in Figure 3.18 a directory has many files. A file may only belong to a single directory.* Within the context of a directory, the file name specifies a unique file. *Directory* and *File* are object classes and *file name* is the qualifier. A directory plus a file name yields a file. A file corresponds to a directory and a file name. Qualification reduces the effective multiplicity of this association from one-to-many to one-to-one. A directory has many files, each with a unique name.

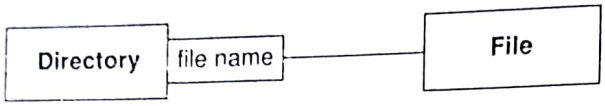


Figure 3.18 A qualified association

Qualification improves semantic accuracy and increases the visibility of navigation paths. It is much more informative to be told that a directory and file name combine to identify a file, rather than be told that a directory has many files. The qualification syntax also indicates that each file name is unique within its directory. One way to find a file is to first find the directory and then traverse the file name link.

A qualifier is drawn as a small box on the end of the association line near the class it qualifies. *Directory* + *file name* yields a *File*, therefore *file name* is listed in a box contiguous to *Directory*.

Qualification often occurs in real problems, frequently because of the need to supply names. There normally is a context within which a name has meaning. For instance, a directory provides the context for a file name.

Figure 3.19 provides another example of qualification. A stock exchange lists many companies. However, a stock exchange lists only one company with a given ticker symbol. A company may be listed on many stock exchanges, possibly under different symbols. (This may actually not be true for stocks.) The unqualified notation cannot accommodate different ticker symbols for the same company on different exchanges.

Qualification usually reduces multiplicity from many to one, but not always. In Figure 3.20, a company has one president and one treasurer but many persons serving on the board of directors. Qualification partitions a set of related objects into disjoint subsets, but the subsets may contain more than one object.

3.3.6 Aggregation

Aggregation is the "part-whole" or "a-part-of" relationship in which objects representing the *components* of something are associated with an object representing the entire *assembly*. One

* This is only true for some operating systems. For example, a PC-DOS file does belong to a single directory. A UNIX file may belong to multiple directories. Once again, the precise nature of an object model depends upon the application.

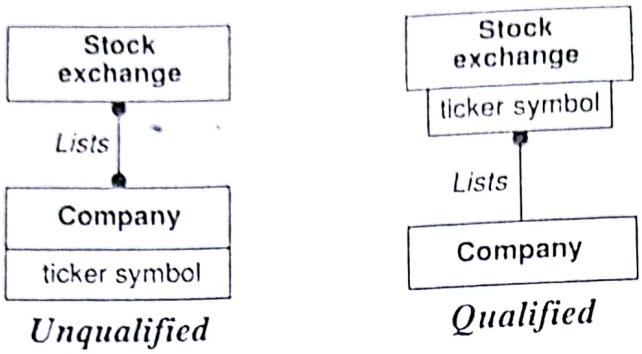
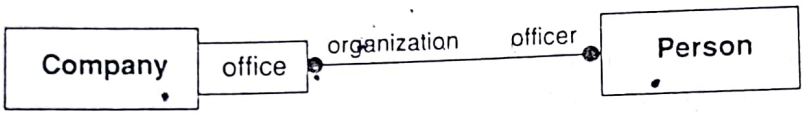


Figure 3.19 Unqualified and qualified association



ABC widgets	President	Roger Slick
ABC widgets	Treasurer	Joe Embezzle
ABC widgets	Director	Joe Doe
ABC widgets	Director	Jane Doe
ABC widgets	Director	Moe Brown
XYZ candy	President	Moe Brown

Figure 3.20 Many-to-many qualification

common example is the bill-of-materials or parts explosion tree. For example, a name, argument list, and a compound statement are part of a C-language function definition, which in turn is part of an entire program. Aggregation is a tightly coupled form of association with some extra semantics. The most significant property of aggregation is *transitivity*, that is, if A is part of B and B is part of C, then A is part of C. Aggregation is also *antisymmetric*, that is, if A is part of B, then B is not part of A. Finally, some properties of the assembly *propagate* to the components as well, possibly with some local modifications. For example, the environment of a statement within a function definition is the same as the environment of the whole function, except for changes made within the function. The speed and location of a door handle is obtained from the door of which it is a part; the door in turn obtains its properties from the car of which it is a part. Unless there are common properties of components that can be attached to the assembly as a whole, there is little point in using aggregation. A parts tree is clearly an aggregation, but there are borderline cases where the use of aggregation is not clear-cut. When in doubt, use ordinary association. Section 4.1 explores the use of aggregation in more detail.

We define an aggregation relationship as relating an assembly class to *one* component class. An assembly with many kinds of components corresponds to many aggregation relationships. 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.

Aggregation is drawn like association, except a small diamond indicates the assembly end of the relationship. Figure 3.21 shows a portion of an object model for a word processing program. A document consists of many paragraphs, each of which consists of many sentences.

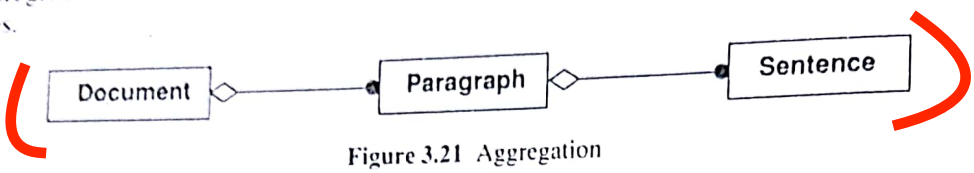


Figure 3.21 Aggregation

The existence of a component object may depend on the existence of the aggregate object of which it is part. For example, a binding is a part of a book. A binding cannot exist apart from a book. In other cases, component objects have an independent existence, such as mechanical parts from a bin.

Figure 3.22 demonstrates that aggregation may have an arbitrary number of levels. A microcomputer is composed of one or more monitors, a system box, an optional mouse, and a keyboard. A system box, in turn, has a chassis, a CPU, many RAM chips, and an optional fan. When we have a collection of components that all belong to the same assembly, we can combine the lines into a single aggregation tree in the diagram. The aggregation tree is just a shorthand notation that is simpler than drawing many lines connecting components to an assembly. An object model should make it easy to visually identify levels in a part hierarchy.

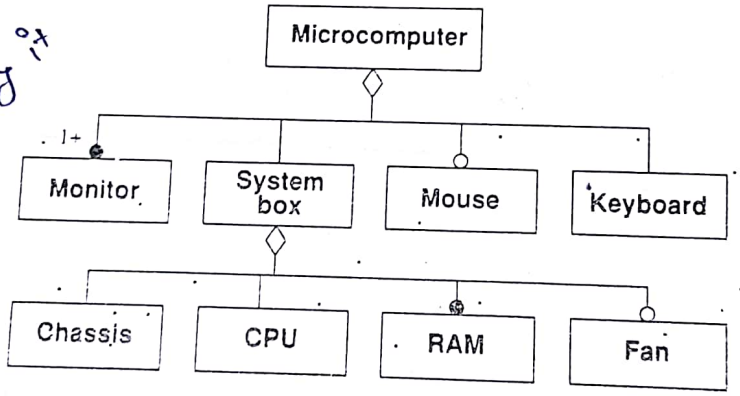


Figure 3.22 Multilevel aggregation

3.4 GENERALIZATION AND INHERITANCE

3.4.1 General Concepts

Generalization and inheritance are powerful abstractions for sharing similarities among classes while preserving their differences. For example, we would like to be able to model the following situation: Each piece of equipment has a manufacturer, weight, and cost.

Pumps also have suction pressure and flow rate. Tanks also have volume and pressure. We would like to define equipment features just once and then add details for pump, tank, and other equipment types.

Generalization is the relationship between a class and one or more refined versions of it. The class being refined is called the *superclass* and each refined version is called a *subclass*. For example, *Equipment* is the superclass of *Pump* and *Tank*. Attributes and operations common to a group of subclasses are attached to the superclass and shared by each subclass. Each subclass is said to *inherit* the features of its superclass. For example, *Pump* inherits attributes *manufacturer*, *weight*, and *cost* from *Equipment*. Generalization is sometimes called the "is-a" relationship because each instance of a subclass is an instance of the superclass as well.

Generalization and inheritance are transitive across an arbitrary number of levels. The terms *ancestor* and *descendent* refer to generalization of classes across multiple levels. An instance of a subclass is simultaneously an instance of all its ancestor classes. The state of an instance includes a value for every attribute of every ancestor class. Any operation on any ancestor class can be applied to an instance. Each subclass not only inherits all the features of its ancestors but adds its own specific attributes and operations as well. For example, *Pump* adds attribute *flow rate*, which is not shared by other kinds of equipment.

The notation for generalization is a triangle connecting a superclass to its subclasses. The superclass is connected by a line to the apex of the triangle. The subclasses are connected by lines to a horizontal bar attached to the base of the triangle. For convenience, the triangle can be inverted, and subclasses can be connected to both the top and bottom of the bar, but if possible the superclass should be drawn on top and the subclasses on the bottom.

Figure 3.23 shows an equipment generalization. Each piece of equipment is a pump, heat exchanger, tank, or another type of equipment. There are several kinds of pumps: centrifugal, diaphragm, and plunger. There are several kinds of tanks: floating roof, pressurized, and spherical. *Pump type* and *tank type* both refine second level generalization classes down to a third level: the fact that the tank generalization symbol is drawn below the pump generalization symbol is not significant. Several object instances are displayed at the bottom of the figure. Each object inherits features from one class at each level of the generalization. Thus *P101* embodies the features of equipment, pump, and diaphragm pump. *E302* assumes the properties of equipment and heat exchanger.

The dangling subclass ellipsis (triple dot) in Figure 3.23 indicates that there are additional subclasses that are not shown on the diagram, perhaps because there is no room on the sheet and they are shown elsewhere, or maybe because enumeration of subclasses is still incomplete.

The words written next to the triangles in the diagram, such as *equipment type*, *pump type*, and *tank type*, are discriminators. A *discriminator* is an attribute of enumeration type that indicates which property of an object is being abstracted by a particular generalization relationship. Only one property should be discriminated at once. For example, class *Vehicle* can be discriminated on propulsion (wind, gas, coal, animal, gravity) and also on operating environment (land, air, water, outer space). The discriminator is simply a name for the basis of generalization. Discriminator values are inherently in one-to-one correspondence with the subclasses of a generalization. For example, the operating environment discriminator for

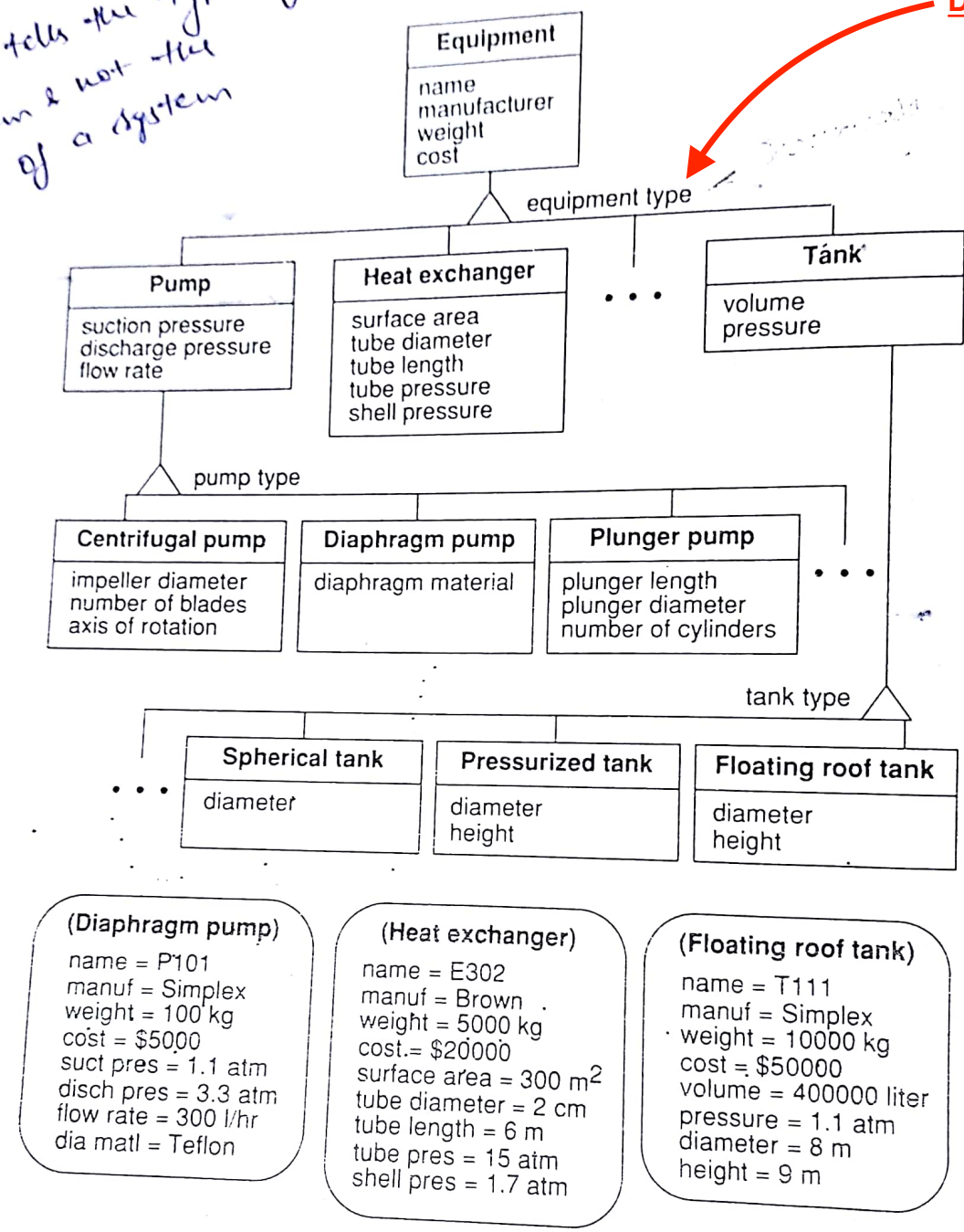


Figure 3.23 A multilevel inheritance hierarchy with instances

Boat is *water*. The discriminator is an optional part of a generalization relationship: if a discriminator is included, it should be drawn next to the generalization triangle.

Figure 3.24 shows classes of graphic geometric figures. This example has more of a programming flavor and emphasizes inheritance of operations. *Move*, *select*, *rotate*, and *display*

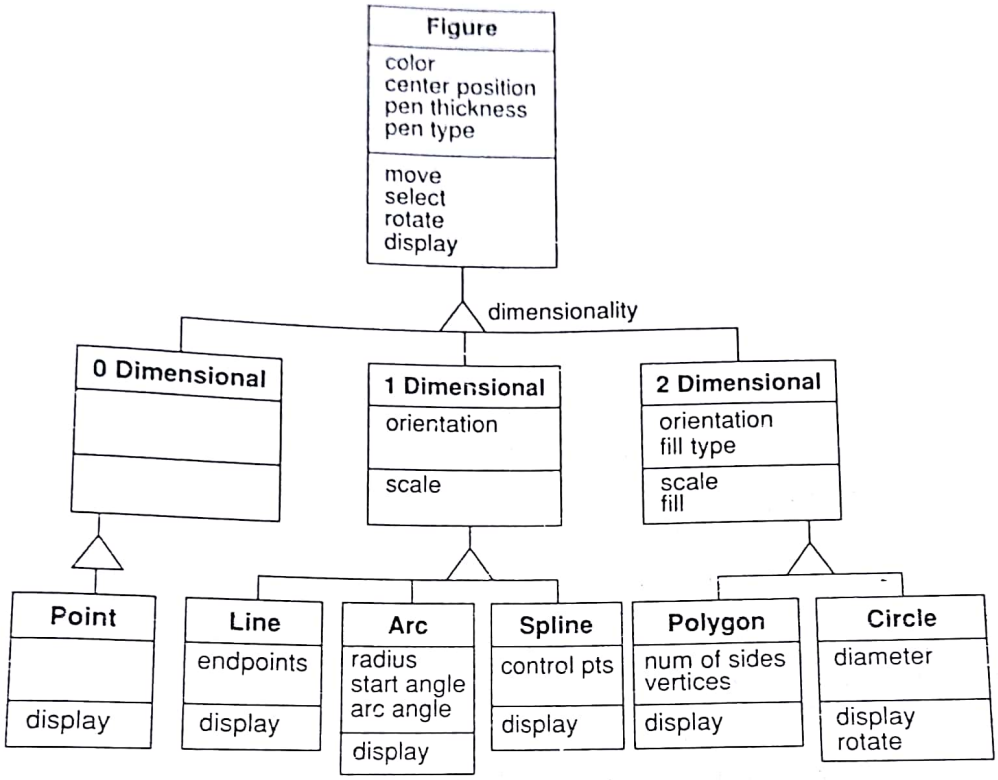


Figure 3.24 Inheritance for graphic figures

are operations inherited by all subclasses. *Scale* applies to one- and two-dimensional figures. *Fill* applies only to two-dimensional figures.

Do not nest subclasses too deeply. Deeply nested subclasses can be difficult to understand, much like deeply nested blocks of code in a procedural language. Often with some careful thought and a little restructuring, you can reduce the depth of an overextended inheritance hierarchy. In practice, whether or not a subclass is "too deeply nested" depends upon judgment and the particular details of a problem. The following guidelines may help: An inheritance hierarchy that is two or three levels deep is certainly acceptable; ten levels deep is probably excessive; five or six levels may or may not be proper.

3.4.2 Use of Generalization

Generalization is a useful construct for both conceptual modeling and implementation. Generalization facilitates modeling by structuring classes and succinctly capturing what is similar and what is different about classes. Inheritance of operations is helpful during implementation as a vehicle for reusing code.

Object-oriented languages provide strong support for the notion of inheritance. (The concept of inheritance was actually invented far earlier, but object-oriented languages made it popular.) In contrast, current database systems provide little or no support for inheritance. Object-oriented database programming languages (Section 15.8.5) and extended relational database systems (Section 17.4) show promise of correcting this situation.

Inheritance has become synonymous with code reuse within the object-oriented programming community. After modeling a system, the developer looks at the resulting classes and tries to group similar classes together and reuse common code. Often code is available from past work (such as a class library) which the developer can reuse and modify, where necessary, to get the precise desired behavior. The most important use of inheritance, however, is the conceptual simplification that comes from reducing the number of independent features in a system.

The terms inheritance, generalization, and specialization all refer to aspects of the same idea and are often used interchangeably. We use *generalization* to refer to the relationship among classes, while *inheritance* refers to the mechanism of sharing attributes and operations using the generalization relationship. Generalization and specialization are two different viewpoints of the same relationship, viewed from the superclass or from the subclasses. The word *generalization* derives from the fact that the superclass generalizes the subclasses. *Specialization* refers to the fact that the subclasses refine or specialize the superclass. In practice, there is little danger of confusion.

3.4.3 Overriding Features

A subclass may *override* a superclass feature by defining a feature with the same name. The overriding feature (the subclass feature) refines and replaces the overridden feature (the superclass feature). There are several reasons why you may wish to override a feature: to specify behavior that depends on the subclass, to tighten the specification of a feature, or for better performance. For example, in Figure 3.24, *display* must be implemented separately for each kind of figure, although it is defined for any kind of figure. Operation *rotate* is overridden for performance in class *Circle* to be a null operation. Chapter 4 discusses overriding features in more detail.

You may override default values of attributes and methods of operations. You should never override the *signature*, or form, of a feature. An override should preserve attribute type, number, and type of arguments to an operation and operation return type. Tightening the type of an attribute or operation argument to be a subclass of the original type is a form of *restriction* (Section 4.3) and must be done with care. It is common to boost performance by overriding a general method with a special method that takes advantage of specific information but does not alter the operation semantics (such as *rotate-circle* in Figure 3.24).

A feature should never be overridden so that it is inconsistent with the signature or semantics of the original inherited feature. A subclass *is* a special case of its superclass and should be compatible with it in every respect. A common, but unfortunate, practice in object-oriented programming is to "borrow" a class that is similar to a desired class and then modify it by changing and ignoring some of its features, even though the new class is not really a

special case of the original class. This practice can lead to conceptual confusion and hidden assumptions built into programs. (See Section 4.3 for further discussion of overrides.)

3.5 GROUPING CONSTRUCTS

3.5.1 Module

A *module* is a logical construct for grouping classes, associations, and generalizations. A module captures one perspective or view of a situation. For example, electrical, plumbing, and ventilation modules are different views of a building. The boundaries of a module are somewhat arbitrary and subject to judgment.

An object model consists of one or more modules. Modules enable you to partition an object model into manageable pieces. Modules provide an intermediate unit of packaging between an entire object model and the basic building blocks of class and association. Class names and association names must be unique within a module. As much as possible, you should use consistent class and association names across modules. The module name is usually listed at the top of each sheet. There is no other special notation for modules.

The same class may be referenced in different modules. In fact, referencing the same class in multiple modules is the mechanism for binding modules together. There should be fewer links between modules (external binding) than within modules (internal binding).

3.5.2 Sheet

A complex model will not fit on a single piece of paper. A *sheet* is the mechanism for breaking a large object model down into a series of pages. A sheet is a single printed page. Each module consists of one or more sheets. As a rule, we never put more than one module per sheet. A sheet is just a notational convenience, not a logical construct.

Each sheet has a title and a name or number. Each association and generalization appears on a single sheet. Classes may appear on multiple sheets. Multiple copies of the same class form the bridge for connecting sheets in an object model. Sheet numbers or sheet names inside circles contiguous to a class box indicate other sheets that refer to a class. Use of sheet cross-reference circles is optional.

3.6 A SAMPLE OBJECT MODEL

Figure 3.25 shows an object model of a workstation window management system, such as the X Window System or SunView. This model is greatly simplified—a real model of a windowing system would require a number of pages—but it illustrates many object modeling constructs and shows how they fit together into a large model.

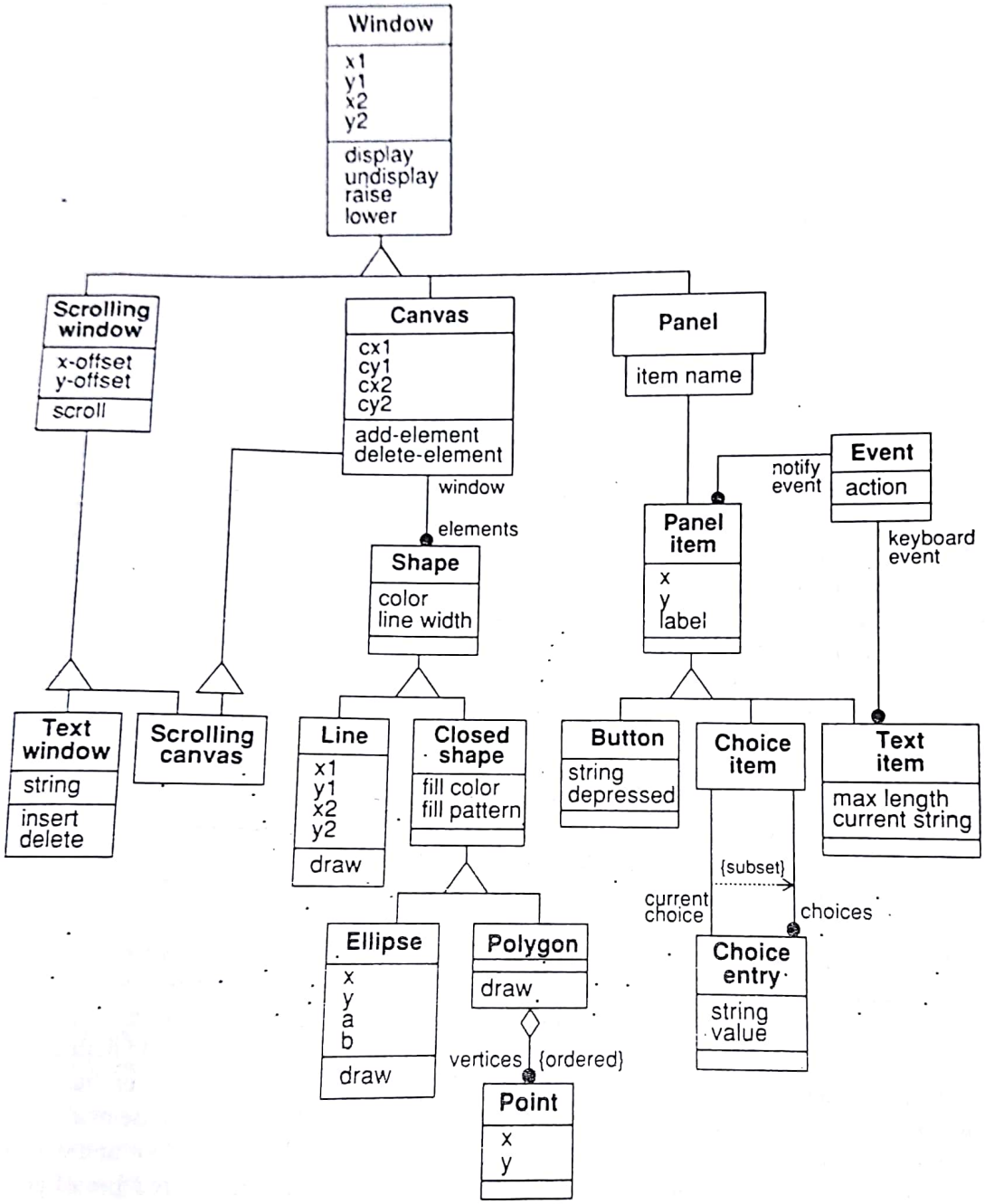


Figure 3.25 Object model of windowing system

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

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

Text window is a kind of a *Scrolling window*, which has a 2-dimensional scrolling offset within its window, as specified by *x-offset* and *y-offset*, as well as an operation *scroll* to change the scroll value. A text window contains a string, and has operations to insert and delete characters. *Scrolling canvas* is a special kind of canvas that supports scrolling; it is both a *Canvas* and a *Scrolling window*. This is an example of *multiple inheritance*, to be explained in Section 4.4.

A *Panel* contains a set of *Panel item* objects, each identified by a unique *item name* within a given panel, as shown by the qualified association. Each panel item belongs to a single panel. A panel item is a predefined icon with which a user can interact on the screen. Panel items come in three kinds: buttons, choice items, and text items. A button has a string which appears on the screen; a button can be pushed by the user and has an attribute *depressed*. A choice item allows the user to select one of a set of predefined *choices*, each of which is a *Choice entry* containing a string to be displayed and a value to be returned if the entry is selected. There are two associations between *Choice item* and *Choice entry*; a one-to-many association defines the set of allowable choices, while a one-to-one association identifies the current choice. The current choice must be one of the allowable choices, so one association is a subset of the other as shown by the arrow between them labeled "{subset}." This is an example of a constraint, to be explained in Section 4.7.

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

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

3.7 PRACTICAL TIPS

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

- Don't begin constructing an object model by merely jotting down classes, associations, and inheritance. First, you must understand the problem to be solved. The content of an object model is driven by relevance to the solution.
- Strive to keep your model simple. Avoid needless complications.
- Carefully choose names. Names are important and carry powerful connotations. Names should be descriptive, crisp, and unambiguous. Names should not be biased towards one aspect of an object. Choosing good names is one of the most difficult aspects of object modeling.
- Do not bury pointers or other object references inside objects as attributes. Instead model these as associations. This is clearer and captures the true intent rather than an implementation approach.
- Try to avoid general ternary and n-ary associations. Most of these can be decomposed into binary associations, with possible qualifiers and link attributes.
- Don't try to get multiplicity perfect too early in software development.
- Do not collapse link attributes into a class.
- Use qualified associations where possible.
- Try to avoid deeply nested generalizations.
- Challenge one-to-one associations. Often the object on either end is optional and zero-or-one multiplicity may be more appropriate. Other times many multiplicity is needed.
- Don't be surprised if your object model requires revision. Object models often require multiple iterations to clarify names, repair errors, add details, and correctly capture structural constraints (Section 4.7). Some of our most complex models, which are only a few pages long, have required half a dozen iterations.
- Try to get others to review your model. Object models can be a focal point for stimulating the involvement of others.
- Always document your object models. The diagram specifies the structure of a model but cannot describe the reasons behind it. The written explanation guides the reader through the model and explains subtle reasons why the model was structured a particular way. The written explanation clarifies the meaning of names in the model and should convey the reason for each class and relationship.
- Do not feel bound to exercise all object modeling constructs. The OMT notation is an idealization. Not all constructs are needed for every problem. Many constructs are optional and a matter of taste. Use only what you need for the problem at hand.

3.8 CHAPTER SUMMARY

Object models describe the static data structure of objects, classes, and their relationships to one another. The content of an object model is a matter of judgment and is driven by its relevance to an application. An object is a concept, abstraction, or thing with crisp boundaries and meaning for an application. All objects have identity and are distinguishable. An object class describes a group of objects with common attributes, operations, and semantics. An attribute is a property of the objects in a class; an operation is an action that may be applied to objects in a class.

Links and associations establish relationships among objects and classes. A link connects two or more objects. An association describes a group of links with common structure and common semantics. Multiplicity specifies how many instances of one class may relate to each instance of another class. An association is a logical construct, of which a pointer is an implementation alternative. There are other ways of implementing associations besides using pointers.

Additional constructs for modeling associations include: link attribute, role, qualifier, and aggregation. A link attribute is a property of the links in an association. Many-to-many associations demonstrate the most compelling rationale for link attributes. Such an attribute is unmistakably a property of the link and cannot be attached to either object. A role is a direction across an association. Roles are particularly useful in dealing with associations between objects of the same class. A qualifier reduces the effective multiplicity of an association by selecting among the set of objects at the many end. Names are often qualifiers. Aggregation is a tightly coupled form of association with special semantics, such as transitive closure and attribute value propagation. Aggregation is commonly encountered in bill-of-material or parts explosion problems.

Generalization and inheritance are fundamental concepts in object-oriented languages that are missing in conventional languages and databases. Generalization is a useful construct for both conceptual modeling and implementation. During conceptual modeling, generalization enables the developer to organize classes into a hierarchical structure based on their similarities and differences. During implementation, inheritance facilitates code reuse. The term *generalization* refers to the relationship among class; the term *inheritance* refers to the mechanism of obtaining attributes and operations using the generalization structure. Generalization provides the means for refining a superclass into one or more subclasses. The superclass contains features common to all classes; the subclasses contain features specific to each class. Inheritance may occur across an arbitrary number of levels where each level represents one aspect of an object. An object accumulates features from each level of a generalization hierarchy.

Module and sheet are grouping constructs. An object model consists of one or more modules. A module is a logical grouping construct which captures one perspective or view of a situation. Most references to classes lie within modules; a few span modules. Each module has one or more sheets. A sheet is merely a notational convenience for fitting object models onto fixed sized pieces of paper.

The various object modeling constructs work together to describe a complex system precisely, as shown by our example of a model for a windowing system. Once an object model is available, even a simplified one, the model can be compared against knowledge of the real world or the desired application, criticized, and improved.

aggregation	generalization	method	override
association	identity	module	qualification
attribute	inheritance	multiplicity	role
class	instance	object	sheet
discriminator	link	operation	signature
feature	link attribute	ordering	specialization

Figure 3.26 Key concepts for Chapter 3

BIBLIOGRAPHIC NOTES

The object modeling approach described in this book builds on the OMT notation originally proposed in [Loomis-87]. [Blaha-88] extends the OMT notation for the purpose of database design. This book redefines the term *OMT*; in this book OMT does not merely refer to a notation but refers to our entire methodology. Our *object model* is analogous to the *OMT notation* discussed in the papers. This book refines object modeling notation beyond that shown in the papers and sets forth a complete methodology for its use.

The object modeling notation is one of a score of approaches descended from the seminal entity-relationship (ER) model of [Chen-76]. All the descendents attempt to improve on the ER approach. Enhancements to the ER model have been pursued for several reasons. The ER technique has been successful for database modeling and as a result, there has been great demand for additional power. Also, ER modeling only addresses database design and not programming. There are too many extensions to ER for us to discuss them all here. (Chapter 12 discusses some extensions to ER.)

A noteworthy aspect of our approach to object modeling is the emphasis we place on associations. Just as inheritance is useful for conceptual modeling and implementation, so too associations are important for conceptual modeling and implementation. Most existing object-oriented programming languages ([Cox-86], [Goldberg-83], and [Meyer-88]) lack the notion of associations and require the use of pointers. Most database design techniques recognize the importance of associations. [Rumbaugh-87] is the original source of our association ideas. The use of the term *relation* in [Rumbaugh-87] is synonymous with our use of *association* in this book.

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

REFERENCES

[Blaha-88] Michael Blaha, William Premerlani, James Rumbaugh. Relational database design using an object-oriented methodology. *Communications of the ACM 31*, 4 (April 1988) 414-427.

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

[Cox-86] Brad J. Cox. *Object-Oriented Programming*. Reading, Mass.: Addison-Wesley, 1986.

[Goldberg-83] Adele Goldberg, David Robson. *Smalltalk-80: The Language and its Implementation.* Reading, Mass.: Addison-Wesley, 1983.

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

[Loomis-87] Mary E.S. Loomis, Ashwin V. Shah, 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.

[Meyer-88] Bertrand Meyer. *Object-Oriented Software Construction.* Hertfordshire, England: Prentice Hall International, 1988.

[Rumbaugh-87] James E. Rumbaugh. Relations as semantic constructs in an object-oriented language. *OOPSLA'87* as *ACM SIGPLAN 22*, 12 (Dec.1987), 466-481.

EXERCISES

3.1 (2) Prepare a class diagram from the instance diagram in Figure E3.1.

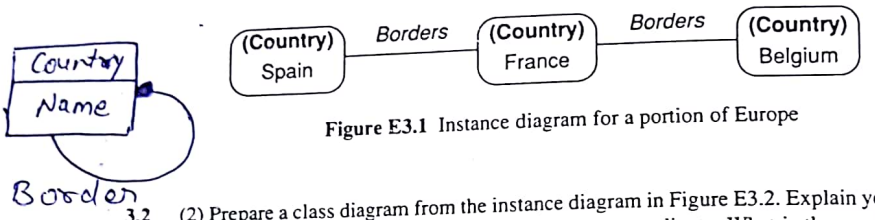


Figure E3.1 Instance diagram for a portion of Europe

3.2 (2) Prepare a class diagram from the instance diagram in Figure E3.2. Explain your multiplicity decisions. Each point has an x coordinate and a y coordinate. What is the smallest number of points required to construct a polygon? Does it make a difference whether or not a given point may be shared between several polygons? How can you express the fact that points are in a sequence?

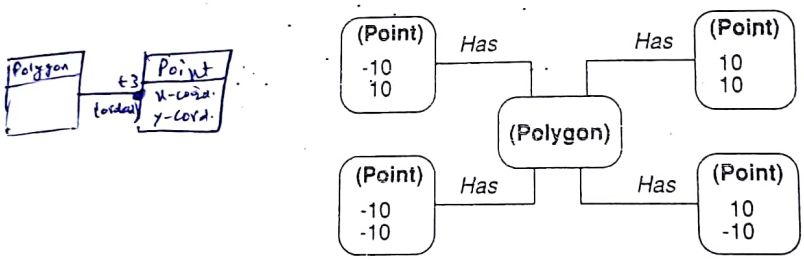
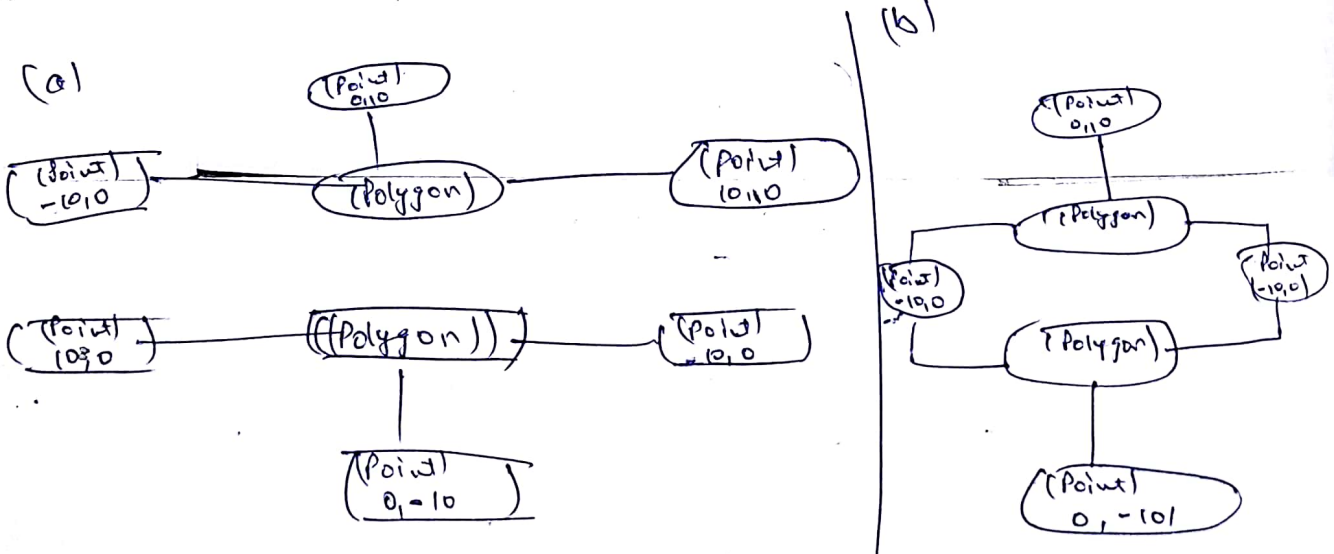


Figure E3.2 Instance diagram of a polygon that happens to be a square

3.3 (3) Consistent with the object diagram that you prepared in exercise 3.2, draw an instance diagram for two triangles with a common side under the following conditions:

a. A point belongs to exactly one polygon.

b. A point belongs to one or more polygons.



3.4 (3) Prepare a class diagram from the instance diagram in Figure E3.3. Explain your multiplicity decisions. How does your diagram express the fact that points are in a sequence?

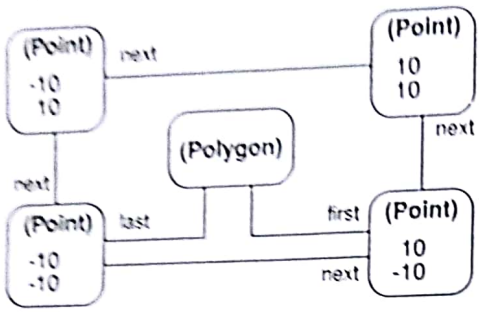


Figure E3.3 Another instance diagram of a polygon that happens to be a square

3.5 (5) Prepare a written description for the object diagrams in exercise 3.2 and exercise 3.4.

3.6 (5) Prepare a class diagram from the instance diagram in Figure E3.4.

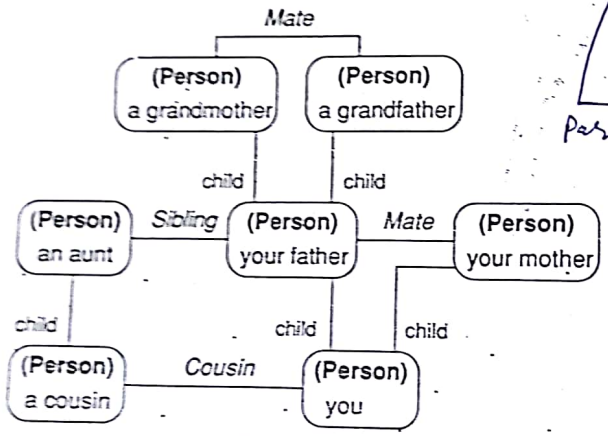


Figure E3.4 Instance-diagram for part of your family tree

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

3.8 (6) a. Prepare an instance diagram for the class diagram in Figure E3.6 for the expression $(X+Y/2)/(X/3+Y)$. Parentheses are used in the expression for grouping, but are not needed in the diagram. The many multiplicity indicates that a term may be used in more than one expression.

b. Modify the class diagram so that terms are not shared and to handle unary minus.

multiplicity

re

se 3.4.

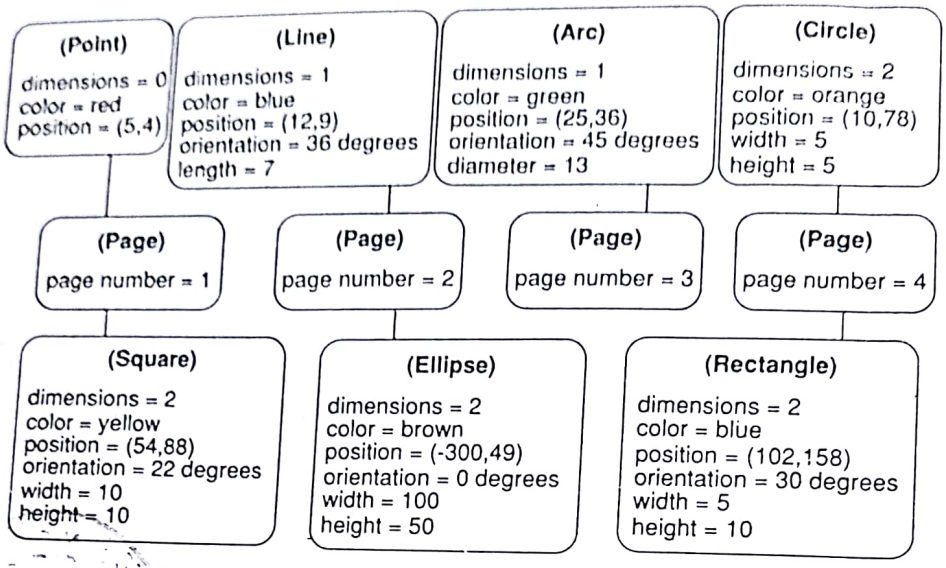


Figure E3.5 Instance diagram for a geometrical document

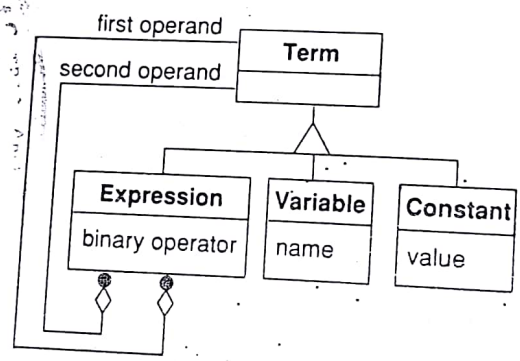


Figure E3.6 Class diagram for simple arithmetic expressions

shown in Fig-
t and a yellow
a circle, and a
ne aggregation

the expression
ıt are not need-
more than one

inus.

3.9 (3) Figure E3.7 is a partially completed object diagram of an air transportation system. Multiplicity balls have been left out. Add them to the diagram. Defend your decisions. Demonstrate how multiplicity decisions depend on your perception of the world.

3.10 (4) Revise Figure E3.7 to make seat location a qualifier.

3.11 (3) Add association names to the unlabeled associations in Figure E3.7.

3.12 (3) Add role names to the unlabeled associations in Figure E3.7.

3.13 (4) Prepare an instance diagram for an imaginary round trip you took last weekend to London. Include at least one instance of each object class. Fortunately, direct flights on a hypersonic

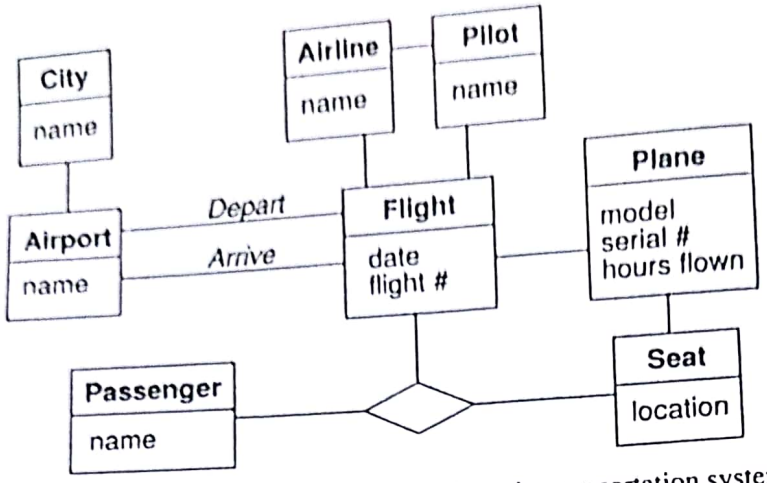


Figure E3.7 Partially completed model of an air transportation system

plane were available. A friend of yours went with you but decided to stay a while and is still there. Captain Johnson was your pilot on both flights. You had a different seat each way, but you noticed it was on the same plane because of a distinctive dent in the tail section.

3.14 (1) Add the following operations to the object diagram in Figure E3.7: heat, hire, fire, refuel, reserve, clean, de-ice, takeoff, land, repair, cancel, delay. It is permissible to add an operation to more than one object class.

3.15 Prepare object diagrams showing at least 10 relationships among the following object classes. Include associations, aggregations, and generalizations. Use qualified associations and show multiplicity balls in your diagrams. You do not need to show attributes or operations. Use association names where needed. As you prepare the diagrams, you may add additional object classes.

a. (4) school, playground, principal, school board, classroom, book, student, teacher, cafeteria, rest room, computer, desk, chair, ruler; door, swing
b. (4) castle, moat, drawbridge, tower, ghost, stairs, dungeon, floor, corridor, room, window, stone, lord, lady, cook
c. (7) expression, constant, variable, function, argument list, relational operator, term, factor, arithmetic operator, statement, program
d. (6) file system, file, directory, file name, ASCII file, executable file, directory file, disk, drive, track, sector
e. (4) automobile, engine, wheel, brake, brake light, door, battery, muffler, tail pipe
f. (6) gas furnace, blower, blower motor, room thermostat, furnace thermostat, humidifier, humidity sensor, gas control, blower control, hot air vents
g. (5) chess piece, rank, file, square, board, move, position, sequence of moves
h. (5) sink, freezer, refrigerator, table, light, switch, window, smoke alarm, burglar alarm, cabinet, bread, cheese, ice, door, kitchen

3.16 (5) Add at least 15 attributes and at least 5 operations to each of the object diagrams you prepared in the previous exercise.

3.17 (5) Figure E3.8 is a portion of an object diagram for a computer program for playing several types of card games. Deck, hand, discard pile, and draw pile are collections of cards. The initial

size of a hand depends on the type of game. Each card has a suit and rank. Add the following operations to the diagram: display, shuffle, deal, initialize, sort, insert, delete, top-of-pile, bottom-of-pile, draw, and discard. Some operations may appear in more than one object class. For each class in which an operation appears, describe the arguments to the operation and what the operation should do to an instance of that class.

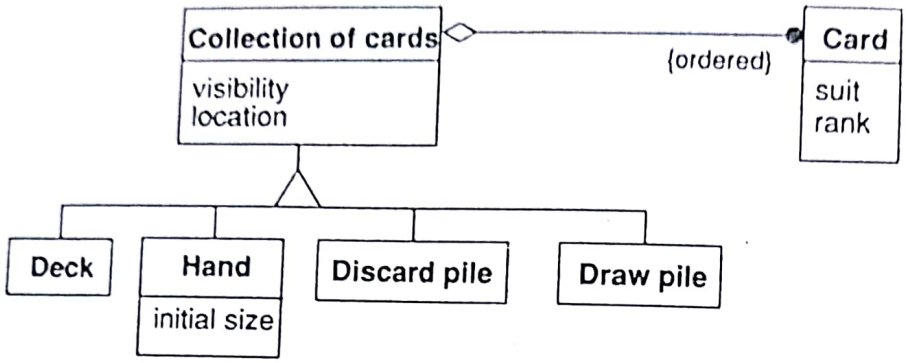


Figure E3.8 Portion of an object diagram for a card playing system

3.18 (5) Figure E3.9 is a portion of an object diagram for a computer system for laying out a newspaper. The system handles several pages which may contain, among other things, columns of text. The user may edit the width and length of a column of text, move it around on a page, or move it from one page to another. As shown, a column is displayed on exactly one page. It is desired to modify the system so that portions of the same column may appear on more than one page. If the user edits the text on one page, the changes should appear automatically on other pages. Modify the object diagram to handle this enhancement. You should change x location and y location into link attributes.

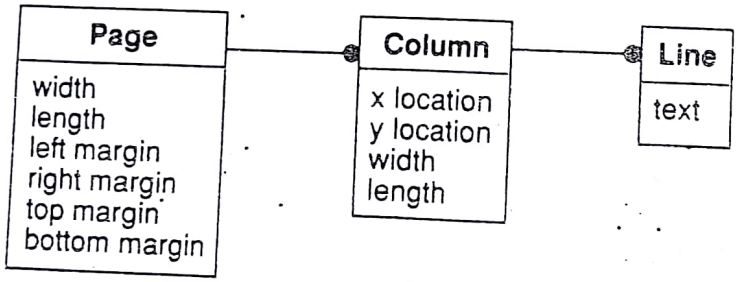


Figure E3.9 Portion of an object diagram for a newspaper publishing system

3.19 (4) Figure E3.10 is an object diagram that might be used in designing a system to simplify the scheduling and scoring of judged athletic competitions such as gymnastics, diving, and figure skating. There are several events and competitors. Each competitor may enter several events and each event has many competitors. Each event has several judges who subjectively rate the performance of competitors in that event. A judge rates every competitor for an event. In some cases, a judge may score more than one event. The focal points of the competition are trials. Each trial is an attempt by one competitor to turn in the best performance possible in one event. A trial is scored by the panel of judges for that event and a net score determined. Add role names and multiplicity balls to the associations.