CHAPTER

20 OBJECT-ORIENTED CONCEPTS AND PRINCIPLES

KEY CONCEPTS

We live in a world of objects. These objects exist in nature, in human-made entities, in business, and in the products that we use. They can be categorized, described, organized, combined, manipulated, and created. Therefore, it is no surprise that an object-oriented view would be proposed for the creation of computer software—an abstraction that enables us to model the world in ways that help us to better understand and navigate it.

An object-oriented approach to the development of software was first proposed in the late 1960s. However, it took almost 20 years for object technologies to become widely used. Throughout the 1990s, object-oriented software engineering became the paradigm of choice for many software product builders and a growing number of information systems and engineering professionals. As time passes, object technologies are replacing classical software development approaches. An important question is why?

The answer (like many answers to questions about software engineering) is not a simple one. Some people would argue that software professionals simply yearned for a "new" approach, but that view is overly simplistic. Object technologies do lead to a number of inherent benefits that provide advantage at both the management and technical levels.

QUICK LOOK

What is it? There are many ways to look at a problem to be solved using a software-based solution. One widely used approach to problem solving takes an object-oriented viewpoint. The problem domain is characterized as a set of objects that have specific attributes and behaviors. The objects are manipulated with a collection of functions (called methods, operations, or services) and communicate with one another through a messaging protocol. Objects are categorized into classes and subclasses.

Who does it? The definition of objects encompasses a description of attributes, behaviors, operations, and messages. This activity is performed by a software engineer.

Why is it important? An object encapsulates both data and the processing that is applied to the data. This important characteristic enables classes of objects to be built and inherently leads to libraries of reusable classes and objects. Because reuse is a critically important attribute of modern software engineering, the object-oriented paradigm is attractive to many software development organizations. In addition, the software components derived using the object-oriented paradigm exhibit design characteristics (e.g., functional independence, information hiding) that are associated with high-quality software.

What are the steps? Object-oriented software engineering follows the same steps as conventional approaches. Analysis identifies objects and classes

QUICK LOOK

that are relevant to the problem domain; design provides the architecture, interface, and component-level detail; implementation (using an object-oriented language) transforms design into code; and testing exercises the object-oriented architecture, interfaces and components.

What is the work product? A set of object oriented models is produced. These models describe the requirements, design, code, and test process for a system or product.

How do I ensure that I've done it right? At each stage, object-oriented work products are reviewed for clarity, correctness, completeness, and consistency with customer requirements and with one another.

Object technologies lead to reuse, and reuse (of program components) leads to faster software development and higher-quality programs. Object-oriented software is easier to maintain because its structure is inherently decoupled. This leads to fewer side effects when changes have to be made and less frustration for the software engineer and the customer. In addition, object-oriented systems are easier to adapt and easier to scale (i.e., large systems can be created by assembling reusable subsystems).

In this chapter we introduce the basic principles and concepts that form a foundation for the understanding of object technologies. Throughout the remainder of Part Four of this book, we consider methods that form the basis for an engineering approach to the creation of object-oriented products and systems.

20.1 THE OBJECT-ORIENTED PARADIGM

For many years, the term *object oriented* (OO) was used to denote a software development approach that used one of a number of object-oriented programming languages (e.g., Ada95, Java, C++, Eiffel, Smalltalk). Today, the OO paradigm encompasses a complete view of software engineering. Edward Berard notes this when he states [BER93]:

Quote:

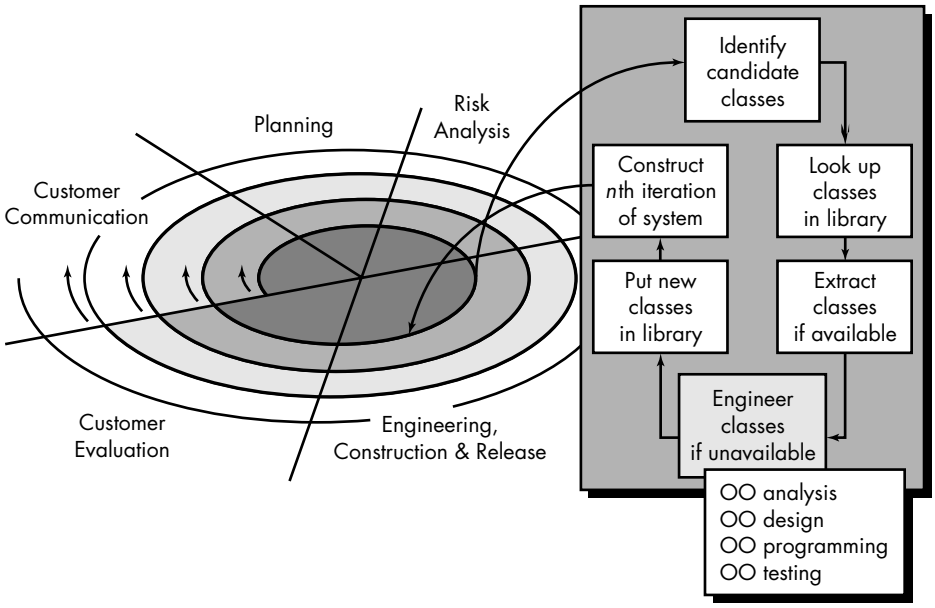
"With objects, it's actually easier to build models [for complex systems] than to engage in sequential programming."

David Taylor

> The benefits of object-oriented technology are enhanced if it is addressed early-on and throughout the software engineering process. Those considering object-oriented technology must assess its impact on the entire software engineering process. Merely employing object-oriented programming (OOP) will not yield the best results. Software engineers and their managers must consider such items as object-oriented requirements analysis (OORA), object-oriented design (OOD), object-oriented domain analysis (OODA), object-oriented database systems (OODBMS) and object-oriented computer aided software engineering (OOCASE).

A reader who is familiar with the conventional approach to software engineering (presented in Part Three of this book) might react to this statement with a shrug: "What's the big deal? We use analysis, design, programming, testing, and related tech-

FIGURE 20.1
The OO process model



nologies when we engineer software using the classical methods. Why should OO be any different?" Indeed, why should OO be any different? In short, it shouldn't!

KEY POINT
OO systems are engineered using an evolutionary process model. Later in this chapter, it will be referred to as a *recursive parallel model.*

In Chapter 2, we discussed a number of different process models for software engineering. Although any one of these models could be adapted for use with OO, the best choice would recognize that OO systems tend to evolve over time. Therefore, an evolutionary process model, coupled with an approach that encourages component assembly (reuse), is the best paradigm for OO software engineering. Referring to Figure 20.1, the component-based development process model (Chapter 2) has been tailored for OO software engineering.

The OO process moves through an evolutionary spiral that starts with customer communication. It is here that the problem domain is defined and that basic problem classes (discussed later in this chapter) are identified. Planning and risk analysis establish a foundation for the OO project plan. The technical work associated with OO software engineering follows the iterative path shown in the shaded box. OO software engineering emphasizes reuse. Therefore, classes are "looked up" in a library (of existing OO classes) before they are built. When a class cannot be found in the library, the software engineer applies object-oriented analysis (OOA), object-oriented design (OOD), object-oriented programming (OOP), and object-oriented testing (OOT) to create the class and the objects derived from the class. The new class is then put into the library so that it may be reused in the future.

WebRef
One of the Web's most extensive lists of OO resources can be found at **mini.net/cetus/software.html**

The object-oriented view demands an evolutionary approach to software engineering. As we will see throughout this and the following chapters, it would be

exceedingly difficult to define all necessary classes for a major system or product in a single iteration. As the OO analysis and design models evolve, the need for additional classes becomes apparent. It is for this reason that the paradigm just described works best for OO.

20.2 OBJECT-ORIENTED CONCEPTS

Quote:

"Object-oriented programming is not so much a coding technique as it is a code packaging technique, a way for code suppliers to encapsulate functionality for delivery to customers."

Brad Cox

Any discussion of object-oriented software engineering must begin by addressing the term *object-oriented.* What is an object-oriented viewpoint? Why is a method considered to be object-oriented? What is an object? Over the years, there have been many different opinions (e.g., [BER93], [TAY90], [STR88], [BOO86]) about the correct answers to these questions. In the discussion that follows, we attempt to synthesize the most common of these.

To understand the object-oriented point of view, consider an example of a real world object—the thing you are sitting in right now—a chair. **Chair** is a member (the term *instance* is also used) of a much larger class of objects that we call **furniture**. A set of generic attributes can be associated with every object in the class **furniture**. For example, all furniture has a cost, dimensions, weight, location, and color, among many possible *attributes.* These apply whether we are talking about a table or a chair, a sofa or an armoire. Because **chair** is a member of **furniture, chair** *inherits* all attributes defined for the class. This concept is illustrated schematically in Figure 20.2.

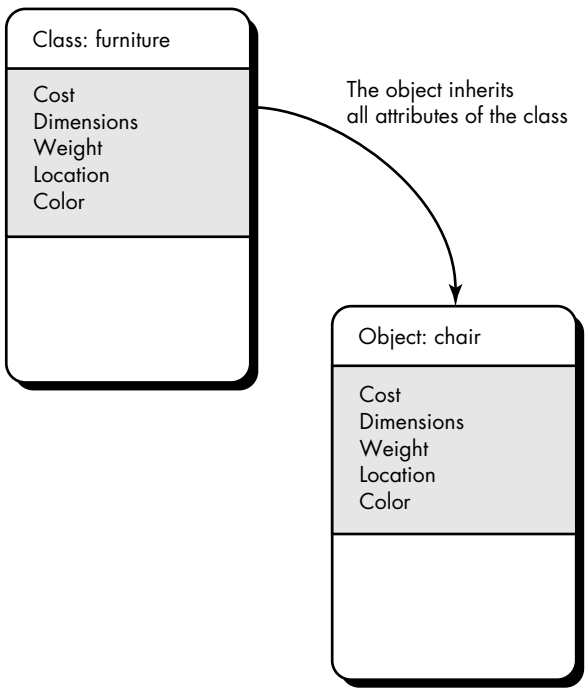


FIGURE 20.2
Inheritance from class to object

Once the class has been defined, the attributes can be reused when new instances of the class are created. For example, assume that we were to define a new object called a **chable** (a cross between a chair and a table) that is a member of the class **furniture. Chable** inherits all of the attributes of **furniture**.

We have attempted an anecdotal definition of a class by describing its attributes, but something is missing. Every object in the class **furniture** can be manipulated in a variety of ways. It can be bought and sold, physically modified (e.g., you can saw off a leg or paint the object purple) or moved from one place to another. Each of these *operations* (other terms are *services* or *methods*) will modify one or more attributes of the object. For example, if the attribute **location** is a composite data item defined as

XRef

Data modeling notation can be used to represent objects and their attributes. See Chapter 12 for details.

location = building + floor + room

then an operation named *move* would modify one or more of the data items (**building, floor,** or **room**) that form the attribute **location**. To do this, *move* must have "knowledge" of these data items. The operation *move* could be used for a chair or a table, as long as both are instances of the class **furniture**. All valid operations (e.g., *buy, sell, weigh*) for the class **furniture** are "connected" to the object definition as shown in Figure 20.3 and are inherited by all instances of the class.

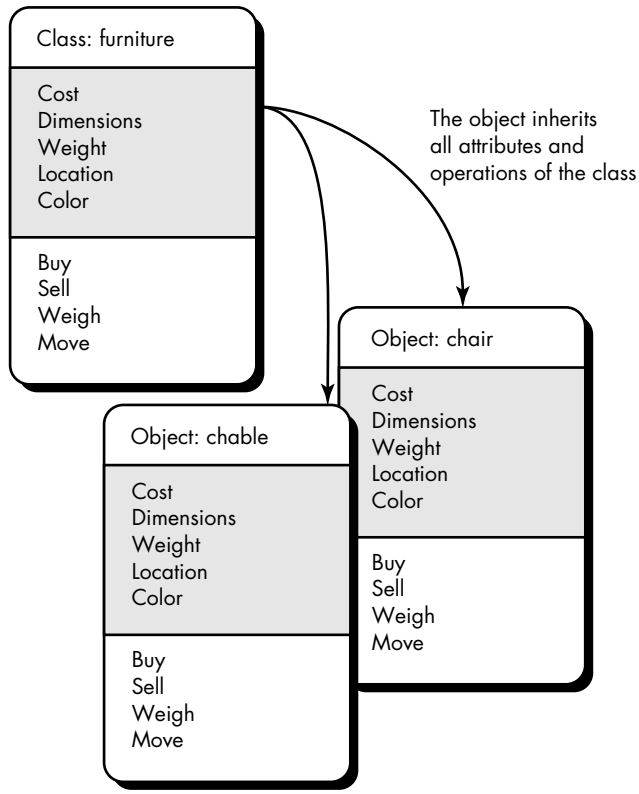


FIGURE 20.3
Inheritance of operations from class to object

Quote:
"Encapsulation prevents a program from becoming so interdependent that a small change has massive ripple effects."
Jim Rumbaugh et al.

The object **chair** (and all objects in general) encapsulates data (the attribute values that define the chair), operations (the actions that are applied to change the attributes of chair), other objects (composite objects can be defined [EVB89]), constants (set values), and other related information. *Encapsulation* means that all of this information is packaged under one name and can be reused as one specification or program component.

Now that we have introduced a few basic concepts, a more formal definition of object-oriented will prove more meaningful. Coad and Yourdon [COA91] define the term this way:

object-oriented = objects + classification + inheritance + communication

Three of these concepts have already been introduced. We postpone a discussion of communication until later.

20.2.1 Classes and Objects

The fundamental concepts that lead to high-quality design (Chapter 13) apply equally to systems developed using conventional and object-oriented methods. For this reason, an OO model of computer software must exhibit data and procedural abstractions that lead to effective modularity. A class is an OO concept that encapsulates the data and procedural abstractions required to describe the content and behavior of some real world entity. Taylor {TAY90] uses the notation shown on the right side of Figure 20.4 to describe a class (and objects derived from a class).

KEY POINT
An object encapsulates both data (attributes) and the functions (operation, methods, or services) that manipulate the data.

The data abstractions (attributes) that describe the class are enclosed by a "wall" of procedural abstractions (called *operations, methods,* or *services*) that are capable of manipulating the data in some way. The only way to reach the attributes (and operate on them) is to go through one of the methods that form the wall. Therefore, the class encapsulates data (inside the wall) and the processing that manipulates the data (the methods that make up the wall). This achieves information hiding and reduces

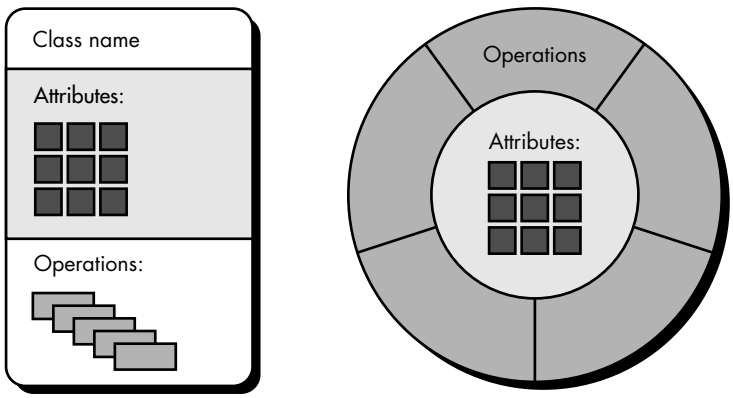


FIGURE 20.4
An alternative representation of an object-oriented class

the impact of side effects associated with change. Since the methods tend to manipulate a limited number of attributes, they are cohesive; and because communication occurs only through the methods that make up the "wall," the class tends to be decoupled from other elements of a system. All of these design characteristics lead to high-quality software.

One of the first things to think about when building an OO system is how to classify the objects that are to be manipulated by the system.

Stated another way, a class is a *generalized description* (e.g., a template, pattern, or blueprint) that describes a collection of similar objects. By definition, all objects that exist within a class inherit its attributes and the operations that are available to manipulate the attributes. A *superclass* is a collection of classes, and a *subclass* is a specialized instance of a class.

These definitions imply the existence of a class hierarchy in which the attributes and operations of the superclass are inherited by subclasses that may each add additional "private" attributes and methods. A class hierarchy for the class **furniture** is illustrated in Figure 20.5.

20.2.2 Attributes

We have already seen that attributes are attached to classes and objects, and that they describe the class or object in some way. A discussion of attributes is presented by de Champeaux, Lea, and Favre [CHA93]:

Real life entities are often described with words that indicate stable features. Most physical objects have features such as shape, weight, color, and type of material. People have features including date of birth, parents, name, and eye color. A feature may be seen as a binary relation between a class and a certain domain.

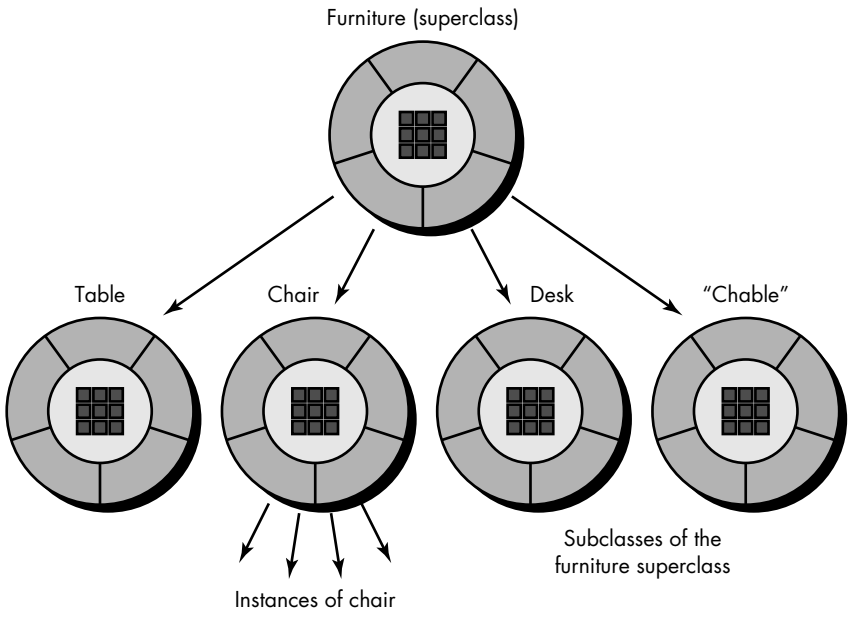


FIGURE 20.5
A class hierarchy

The "binary relation" implies that an attribute can take on a value defined by an enumerated domain. In most cases, a domain is simply a set of specific values. For example, assume that a class **automobile** has an attribute **color**. The domain of values for **color** is {**white, black, silver, gray, blue, red, yellow, green**}. In more complex situations, the domain can be a set of classes. Continuing the example, the class **automobile** also has an attribute **power train** that encompasses the following domain of classes: {**16-valve economy option, 16-valve sport option, 24-valve sport option, 32-valve luxury option**}. Each of the options noted has a set of specific attributes of its own.

The values assigned to an object's attributes make that object unique.

The features (values of the domain) can be augmented by assigning a default value (feature) to an attribute. For example, the **power train** attribute defaults to **16-valve sport option**. It may also be useful to associate a probability with a particular feature by assigned {value, probability} pairs. Consider the **color** attribute for **automobile**. In some applications (e.g., manufacturing planning) it might be necessary to assign a probability to each of the colors (white and black are highly probable as automobile colors).

20.2.3 Operations, Methods, and Services

An object encapsulates data (represented as a collection of attributes) and the algorithms that process the data. These algorithms are called operations, methods, or services[1] and can be viewed as modules in a conventional sense.

Each of the operations that is encapsulated by an object provides a representation of one of the behaviors of the object. For example, the operation *GetColor* for the object **automobile** will extract the color stored in the **color** attribute. The implication of the existence of this operation is that the class **automobile** has been designed to receive a stimulus [JAC92] (we call the stimulus a *message*) that requests the color of the particular instance of a class. Whenever an object receives a stimulus, it initiates some behavior. This can be as simple as retrieving the color of automobile or as complex as the initiation of a chain of stimuli that are passed among a variety of different objects. In the latter case, consider an example in which the initial stimulus received by object 1 results in the generation of two other stimuli that are sent to object 2 and object 3. Operations encapsulated by the second and third objects act on the stimuli, returning necessary information to the first object. Object 1 then uses the returned information to satisfy the behavior demanded by the initial stimulus.

Whenever an object is stimulated by a message, it initiates some behavior by executing an operation.

20.2.4 Messages

Messages are the means by which objects interact. Using the terminology introduced in the preceding section, a message stimulates some behavior to occur in the receiving object. The behavior is accomplished when an operation is executed.

---

1 In the context of this discussion, we use the term *operations,* but *methods* and *services* are equally popular.

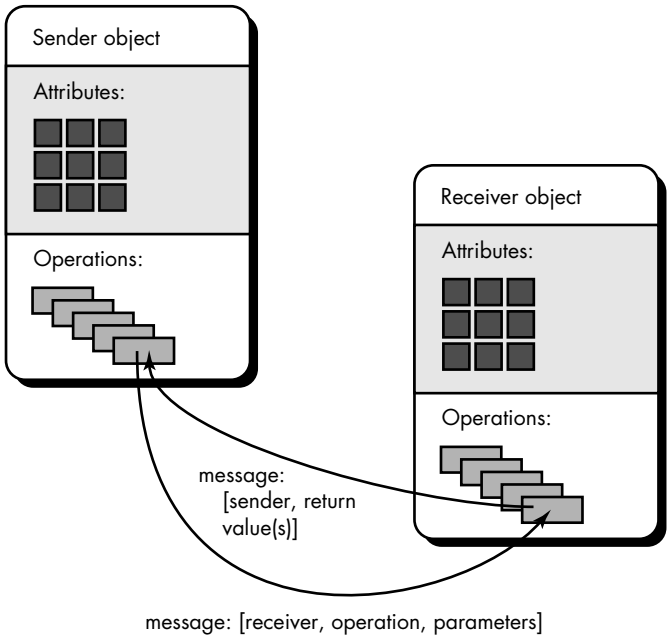


FIGURE 20.6 Message passing between objects

The interaction between messages is illustrated schematically in Figure 20.6. An operation within a sender object generates a message of the form

Message: [destination, operation, parameters]

where *destination* defines the *receiver object* that is stimulated by the message, *operation* refers to the operation that is to receive the message, and *parameters* provides information that is required for the operation to be successful.

Quote:

"Messages and methods [operations] are two sides of the same coin. Methods are the procedures that are invoked when an object receives a message."

Greg Voss

As an example of message passing within an OO system, consider the objects shown in Figure 20.7. Four objects, **A, B, C,** and **D** communicate with one another by passing messages. For example, if object **B** requires processing associated with operation *op10* of object **D**, it would send **D** a message of the form

message: [**D**, *op10,* {data}]

As part of the execution of *op10*, object **D** may send a message to object **C** of the form

message: (**C**, *op08*, {data})

Then **C** finds *op08*, performs it, and sends an appropriate return value to **D**. Operation *op10* completes and sends a return value to **B**.

Cox {COX86] describes the interchange between objects in the following manner:

> An object is requested to perform one of its operations by sending it a message telling the object what to do. The receiver [object] responds to the message by first choosing the operation that implements the message name, executing this operation, and then returning control to the caller.

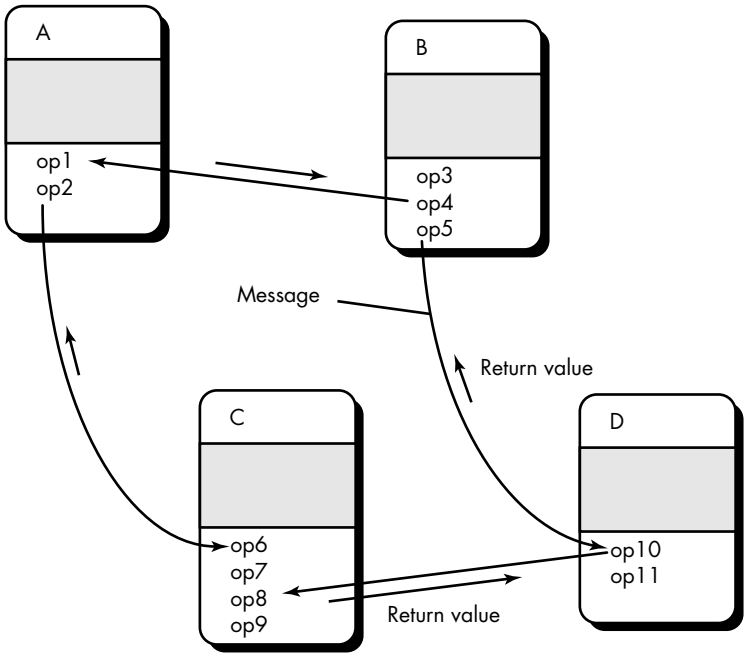


FIGURE 20.7 Message passing example

Messaging ties an object-oriented system together. Messages provide insight into the behavior of individual objects and the OO system as a whole.

20.2.5 Encapsulation, Inheritance, and Polymorphism

Although the structure and terminology introduced in Sections 20.2.1 through 20.2.4 differentiate OO systems from their conventional counterparts, three characteristics of object-oriented systems make them unique. As we have already noted, the OO class and the objects spawned from the class encapsulate data and the operations that work on the data in a single package. This provides a number of important benefits:

? What are the primary benefits of an OO architecture?

- The internal implementation details of data and procedures are hidden from the outside world (information hiding). This reduces the propagation of side effects when changes occur.
- Data structures and the operations that manipulate them are merged in a single named entity—the class. This facilitates component reuse.
- Interfaces among encapsulated objects are simplified. An object that sends a message need not be concerned with the details of internal data structures. Hence, interfacing is simplified and the system coupling tends to be reduced.

Inheritance is one of the key differentiators between conventional and OO systems. A subclass **Y** inherits all of the attributes and operations associated with its superclass, **X**. This means that all data structures and algorithms originally designed

and implemented for **X** are immediately available for **Y**—no further work need be done. Reuse has been accomplished directly.

Any change to the data or operations contained within a superclass is immediately inherited by all subclasses that have inherited from the superclass.[2] Therefore, the class hierarchy becomes a mechanism through which changes (at high levels) can be immediately propagated through a system.

It is important to note that, at each level of the class hierarchy, new attributes and operations may be added to those that have been inherited from higher levels in the hierarchy. In fact, whenever a new class is to be created, the software engineer has a number of options:

- The class can be designed and built from scratch. That is, inheritance is not used.
- The class hierarchy can be searched to determine if a class higher in the hierarchy contains most of the required attributes and operations. The new class inherits from the higher class and additions may then be added, as required.
- The class hierarchy can be restructured so that the required attributes and operations can be inherited by the new class.
- Characteristics of an existing class can be overridden and private versions of attributes or operations are implemented for the new class.

Quote:

"Whereas an object is a concrete entity that exists in time and space, a class represents only an abstraction, the 'essence' of an object, as it were."

Grady Booch

To illustrate how restructuring of the class hierarchy might lead to a desired class, consider the example shown in Figures 20.8. The class hierarchy illustrated in Figure 20.8A enables us to derive classes **X3** and **X4** with characteristics 1, 2, 3, 4, 5 and 6 and 1, 2, 3, 4, 5, and 7, respectively.[3] Now, suppose that a new class with only characteristics 1, 2, 3, 4, and 8 is desired. To derive this class, called **X2b** in the example, the hierarchy may be restructured as shown in Figure 20.8B. It is important to note that restructuring the hierarchy can be difficult, and for this reason, *overriding* is sometimes used.

In essence, overriding occurs when attributes and operations are inherited in the normal manner but are then modified to the specific needs of the new class. As Jacobson notes, when overriding is used "inheritance is not transitive" [JAC92].

In some cases, it is tempting to inherit some attributes and operations from one class and others from another class. This is called *multiple inheritance,* and it is controversial. In general, multiple inheritance complicates the class hierarchy and creates potential problems in configuration control (Chapter 9). Because multiple inheritance sequences are more difficult to trace, changes to the definition of a class that resides high in the hierarchy may have an unintended impact on classes defined lower in the architecture.

---

2 The terms *descendants* and *ancestors* [JAC92] are sometimes used to replace *subclass* and *superclass*, respectively.

3 For the purposes of this example, "characteristics" may be either attributes or operations.

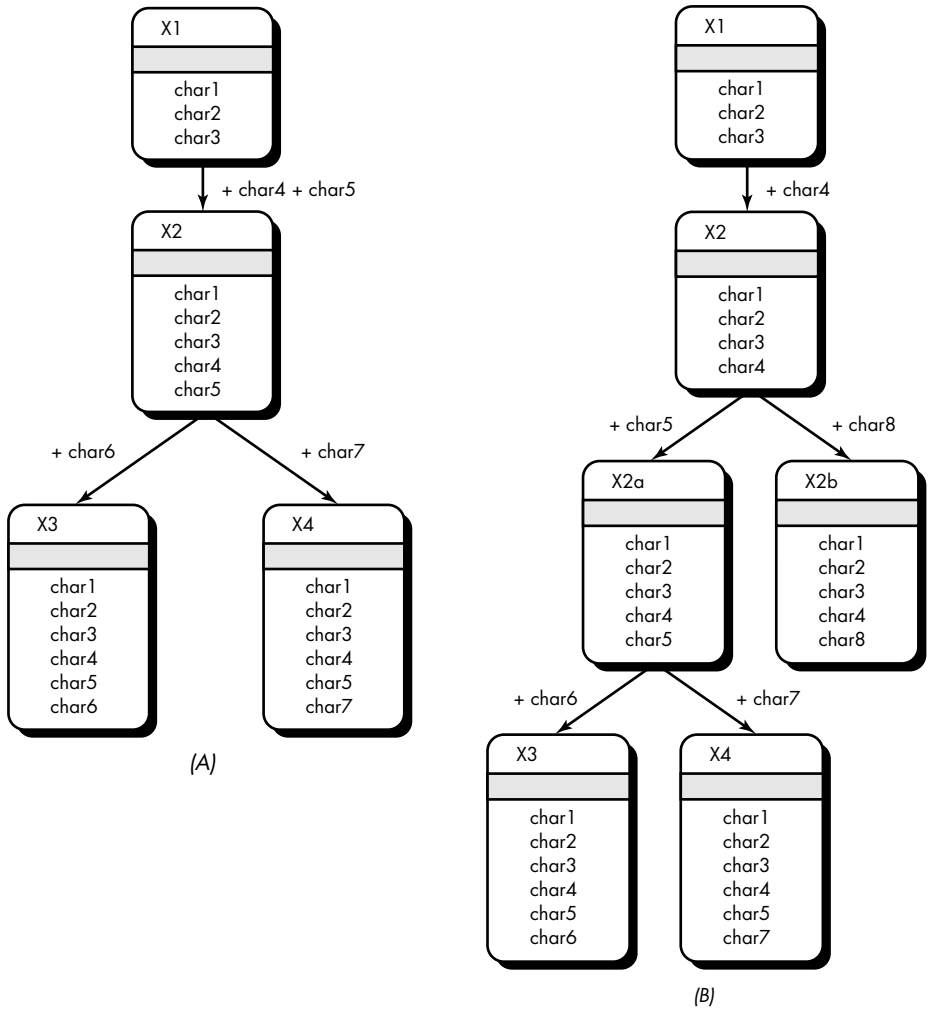


FIGURE 20.8 Class hierarchy: original (A), restructured (B)

Quote:

"The name [polymorphism] may be awkward, but the mechanism is sheer elegance."

David Taylor

Polymorphism is a characteristic that greatly reduces the effort required to extend an existing OO system. To understand polymorphism, consider a conventional application that must draw four different types of graphs: line graphs, pie charts, histograms, and Kiviat diagrams. Ideally, once data are collected for a particular type of graph, the graph should draw itself. To accomplish this in a conventional application (and maintain module cohesion), it would be necessary to develop drawing modules for each type of graph. Then, within the design for each graph type, control logic similar to the following would have to be embedded:

```
case of graphtype:
    if graphtype = linegraph then DrawLineGraph (data);
    if graphtype = piechart then DrawPieChart (data);
```

```
    if graphtype = histogram then DrawHisto (data);
    if graphtype = kiviat then DrawKiviat (data);
end case;
```

Although this design is reasonably straightforward, adding new graph types could be tricky. A new drawing module would have to be created for each graph type and then the control logic would have to be updated for each graph.

To solve this problem, all of the graphs become subclasses of a general class called **graph**. Using a concept called *overloading* [TAY90], each subclass defines an operation called *draw*. An object can send a *draw* message to any one of the objects instantiated from any one of the subclasses. The object receiving the message will invoke its own *draw* operation to create the appropriate graph. Therefore, the design is reduced to

```
graphtype draw
```

When a new graph type is to be added to the system, a subclass is created with its own *draw* operation. But no changes are required within any object that wants a graph drawn because the message **graphtype draw** remains unchanged. To summarize, polymorphism enables a number of different operations to have the same name. This in turn decouples objects from one another, making each more independent.

20.3 IDENTIFYING THE ELEMENTS OF AN OBJECT MODEL

The elements of an object model—classes and objects, attributes, operations, and messages—were each defined and discussed in the preceding section. But how do we go about identifying these elements for an actual problem? The sections that follow present a series of informal guidelines that will assist in the identification of the elements of the object model.

20.3.1 Identifying Classes and Objects

Quote:

"The really hard problem [in OO] is discovering what are the 'right' objects in the first place."

Carl Argila

If you look around a room, there is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations). But when you "look around" the problem space of a software application, the objects may be more difficult to comprehend.

We can begin to identify objects[4] by examining the problem statement or (using the terminology from Chapter 12) by performing a "grammatical parse" on the processing narrative for the system to be built. Objects are determined by underlining each noun or noun clause and entering it in a simple table. Synonyms should be

4 In reality, OOA actually attempts to define classes from which objects are instantiated. Therefore, when we isolate potential objects, we also identify members of potential classes.

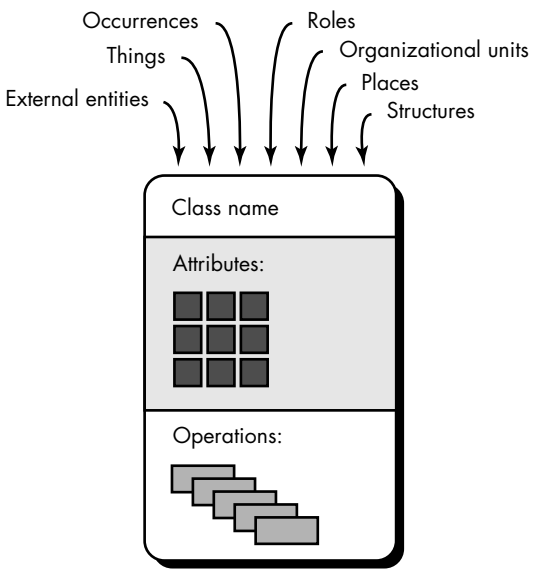


FIGURE 20.9 How objects manifest themselves

noted. If the object is required to implement a solution, then it is part of the solution space; otherwise, if an object is necessary only to describe a solution, it is part of the problem space. What should we look for once all of the nouns have been isolated?

Objects manifest themselves in one of the ways represented in Figure 20.9. Objects can be

? How do I pick out objects as I study the problem to be solved?

- *External entities* (e.g., other systems, devices, people) that produce or consume information to be used by a computer-based system.
- *Things* (e.g, reports, displays, letters, signals) that are part of the information domain for the problem.
- *Occurrences* or *events*[5] (e.g., a property transfer or the completion of a series of robot movements) that occur within the context of system operation.
- *Roles* (e.g., manager, engineer, salesperson) played by people who interact with the system.
- *Organizational units* (e.g., division, group, team) that are relevant to an application.
- *Places* (e.g., manufacturing floor or loading dock) that establish the context of the problem and the overall function of the system.
- *Structures* (e.g., sensors, four-wheeled vehicles, or computers) that define a class of objects or in the extreme, related classes of objects.

5 In this context, the term *event* connotes any occurrence. It does not necessarily imply control as it did in Chapter 12.

This categorization is but one of many that have been proposed in the literature. For example, Budd [BUD96] suggests a taxonomy of classes that includes producers (sources) and consumers (sinks) of data, data managers, view or observer classes, and helper classes.

It is also important to note what objects are not. In general, an object should never have an "imperative procedural name" [CAS89]. For example, if the developers of software for a medical imaging system defined an object with the name **image inversion**, they would be making a subtle mistake. The image obtained from the software could, of course, be an object (it is a thing that is part of the information domain). Inversion of the image is an operation that is applied to the object. It is likely that *inversion* would be defined as an operation for the object **image**, but it would not be defined as a separate object to connote "image inversion." As Cashman [CAS89] states: "the intent of object-orientation is to encapsulate, but still keep separate, data and operations on the data."

To illustrate how objects might be defined during the early stages of analysis, we return to the *SafeHome* security system example. In Chapter 12, we performed a "grammatical parse" on a processing narrative for the *SafeHome* system. The processing narrative is reproduced:

> *SafeHome* software enables the homeowner to configure the security system when it is installed, monitors all sensors connected to the security system, and interacts with the homeowner through a keypad and function keys contained in the *SafeHome* control panel shown in Figure 11.2.
>
> During installation, the *SafeHome* control panel is used to "program" and configure the system. Each sensor is assigned a number and type, a master password is programmed for arming and disarming the system, and telephone number(s) are input for dialing when a sensor event occurs.
>
> When a sensor event is sensed by the software, it rings an audible alarm attached to the system. After a delay time that is specified by the homeowner during system configuration activities, the software dials a telephone number of a monitoring service, provides information about the location, reporting and the nature of the event that has been detected. The number will be redialed every 20 seconds until telephone connection is obtained.
>
> All interaction with *SafeHome* is managed by a user-interaction subsystem that reads input provided through the keypad and function keys, displays prompting messages on the LCD display, displays system status information on the LCD display. Keyboard interaction takes the following form . . .

A grammatical parse can be used to isolate potential objects (nouns) and operations (verbs).

Extracting the nouns, we can propose a number of potential objects:

Potential Object/Class	General Classification
homeowner	role or external entity
sensor	external entity
control panel	external entity
installation	occurrence
system (alias security system)	thing
number, type	not objects, attributes of sensor

Potential Object/Class	General Classification
master password	thing
telephone number	thing
sensor event	occurrence
audible alarm	external entity
monitoring service	organizational unit or external entity

The list would be continued until all nouns in the processing narrative have been considered. Note that we call each entry in the list a *potential* object. We must consider each further before a final decision is made.

How do I know whether a potential object is a good candidate for use in an OO system?

Coad and Yourdon [COA91] suggest six selection characteristics that should be used as an analyst considers each potential object for inclusion in the analysis model:

1. **Retained information.** The potential object will be useful during analysis only if information about it must be remembered so that the system can function.
2. **Needed services.** The potential object must have a set of identifiable operations that can change the value of its attributes in some way.
3. **Multiple attributes.** During requirement analysis, the focus should be on "major" information; an object with a single attribute may, in fact, be useful during design, but is probably better represented as an attribute of another object during the analysis activity.
4. **Common attributes.** A set of attributes can be defined for the potential object and these attributes apply to all occurrences of the object.
5. **Common operations.** A set of operations can be defined for the potential object and these operations apply to all occurrences of the object.
6. **Essential requirements.** External entities that appear in the problem space and produce or consume information essential to the operation of any solution for the system will almost always be defined as objects in the requirements model.

KEY POINT

A potential object should satisfy most or all of these characteristics if it is to be used in the analysis model.

To be considered a legitimate object for inclusion in the requirements model, a potential object should satisfy all (or almost all) of these characteristics. The decision for inclusion of potential objects in the analysis model is somewhat subjective, and later evaluation may cause an object to be discarded or reinstated. However, the first step of OOA must be a definition of objects, and decisions (even subjective ones) must be made. With this in mind, we apply the selection characteristics to the list of potential *SafeHome* objects:

Potential Object/Class	Characteristic Number That Applies
homeowner	rejected: 1, 2 fail even though 6 applies
sensor	accepted: all apply
control panel	accepted: all apply
installation	rejected
system (alias security system)	accepted: all apply

number, type	rejected: 3 fails, attributes of sensor
master password	rejected: 3 fails
telephone number	rejected: 3 fails
sensor event	accepted: all apply
audible alarm	accepted: 2, 3, 4, 5, 6 apply
monitoring service	rejected: 1, 2 fail even though 6 applies

It should be noted that (1) the preceding list is not all-inclusive, additional objects would have to be added to complete the model; (2) some of the rejected potential objects will become attributes for those objects that were accepted (e.g., **number** and **type** are attributes of **sensor**, and **master password** and **telephone number** may become attributes of **system**); (3) different statements of the problem might cause different "accept or reject" decisions to be made (e.g., if each homeowner had an individual password or was identified by voice print, the **homeowner** object would satisfy characteristics 1 and 2 and would have been accepted).

20.3.2 Specifying Attributes

Attributes describe an object that has been selected for inclusion in the analysis model. In essence, it is the attributes that define the object—that clarify what is meant by the object in the context of the problem space. For example, if we were to build a system that tracks baseball statistics for professional baseball players, the attributes of the object **player** would be quite different than the attributes of the same object when it is used in the context of the professional baseball pension system. In the former, attributes such as **name, position, batting average, fielding percentage, years played,** and **games played** might be relevant. For the latter, some of these attributes would be meaningful, but others would be replaced (or augmented) by attributes like **average salary, credit toward full vesting, pension plan options chosen, mailing address,** and the like.

Attributes are chosen by examining the problem statement, looking for things that fully define an object and make it unique.

To develop a meaningful set of attributes for an object, the analyst can again study the processing narrative (or statement of scope) for the problem and select those things that reasonably "belong" to the object. In addition, the following question should be answered for each object: "What data items (composite and/or elementary) fully define this object in the context of the problem at hand?"

To illustrate, we consider the **system** object defined for *SafeHome.* We noted earlier in the book that the homeowner can configure the security system to reflect sensor information, alarm response information, activation/deactivation information, identification information, and so forth. Using the content description notation defined for the data dictionary and presented in Chapter 12, we can represent these composite data items in the following manner:

sensor information = sensor type + sensor number + alarm threshold
alarm response information = delay time + telephone number + alarm type
activation/deactivation information = master password + number of allowable tries + temporary password
identification information = system ID + verification phone number + system status

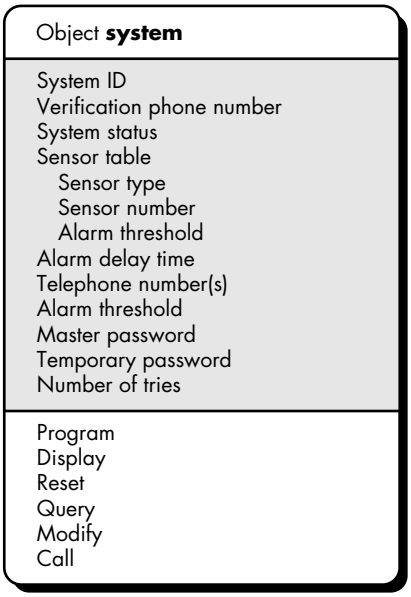


FIGURE 20.10 The **system** object with operations attached

Each of the data items to the right of the equal sign could be further defined to an elementary level, but for our purposes, they constitute a reasonable list of attributes for the system object (shaded portion of Figure 20.10).

20.3.3 Defining Operations

Operations define the behavior of an object and change the object's attributes in some way. More specifically, an operation changes one or more attribute values that are contained within the object. Therefore, an operation must have "knowledge" of the nature of the object's attributes and must be implemented in a manner that enables it to manipulate the data structures that have been derived from the attributes.

? Is there a reasonable way to categorize an object's operations?

Although many different types of operations exist, they can generally be divided into three broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, and (3) operations that monitor an object for the occurrence of a controlling event.

As a first iteration at deriving a set of operations for the objects of the analysis model, the analyst can again study the processing narrative (or statement of scope) for the problem and select those operations that reasonably belong to the object. To accomplish this, the grammatical parse is again studied and verbs are isolated. Some of these verbs will be legitimate operations and can be easily connected to a specific object. For example, from the *SafeHome* processing narrative presented earlier in this chapter, we see that "sensor is assigned a number and type" or that "a master pass-

word is programmed for arming and disarming the system." These two phrases indicate a number of things:

- That an *assign* operation is relevant for the **sensor** object.
- That a *program* operation will be applied to the **system** object.
- That *arm* and *disarm* are operations that apply to **system** (also that **system status** may ultimately be defined (using data dictionary notation) as

  system status = [armed | disarmed]

Upon further investigation, it is likely that the operation *program* will be divided into a number of more specific suboperations required to configure the system. For example, *program* implies specifying phone numbers, configuring system characteristics (e.g., creating the sensor table, entering alarm characteristics), and entering password(s). But for now, we specify *program* as a single operation.

In addition to the grammatical parse, we can gain additional insight into other operations by considering the communication that occurs between objects. Objects communicate by passing messages to one another. Before continuing with the specification of operations, we explore this matter in a bit more detail.

20.3.4 Finalizing the Object Definition

The definition of operations is the last step in completing the specification of an object. In Section 20.3.3, operations were culled from a grammatical parse of the processing narrative for the system. Additional operations may be determined by considering the "life history" [COA91] of an object and the messages that are passed among objects defined for the system.

The generic life history of an object can be defined by recognizing that the object must be created, modified, manipulated or read in other ways, and possibly deleted. For the **system** object, this can be expanded to reflect known activities that occur during its life (in this case, during the time that *SafeHome* is operational). Some of the operations can be ascertained from likely communication between objects. For example, **sensor event** will send a message to **system** to *display* the **event location** and **number**; **control panel** will send **system** a *reset* message to update **system status**; the **audible alarm** will send a *query* message; the **control panel** will send a *modify* message to change one or more attributes without reconfiguring the entire system object; **sensor event** will also send a message to *call* the **phone number(s)** contained in the object. Other messages can be considered and operations derived. The resulting object definition is shown in Figure 20.10.

A similar approach would be used for each of the objects defined for *SafeHome*. After attributes and operations are defined for each of the objects identified to this point, the beginnings of an OOA model would be created. A more detailed discussion of the analysis model that is created as part of OOA is presented in Chapter 21.

20.4 MANAGEMENT OF OBJECT-ORIENTED SOFTWARE PROJECTS

OO projects require as much or more management planning and oversight as conventional software projects. Do not assume that OO somehow relieves you of this responsibility.

As we discussed in Parts One and Two of this book, modern software project management can be subdivided into the following activities:

1. Establishing a common process framework for a project.
2. Using the framework and historical metrics to develop effort and time estimates.
3. Establishing deliverables and milestones that will enable progress to be measured.
4. Defining checkpoints for risk management, quality assurance, and control.
5. Managing the changes that invariably occur as the project progresses.
6. Tracking, monitoring, and controlling progress.

An extensive OO project management tutorial and set of pointers can be found at **mini.net/cetus/oo_project_mngt.html**

The technical manager who is faced with an object-oriented project applies these six activities. But, because of the unique nature of object-oriented software, each of these management activities has a subtly different feel and must be approached using a somewhat different mind-set.

In the sections that follow, we explore software project management for object-oriented projects. The fundamental principles of management stay the same, but the technique must be adapted so that an OO project is properly managed.

20.4.1 The Common Process Framework for OO

A common process framework defines an organization's approach to software engineering. It identifies the paradigm that is applied to build and maintain software and the tasks, milestones, and deliverables that will be required. It establishes the degree of rigor with which different kinds of projects will be approached. The CPF is always adaptable so it can meet the individual needs of a project team. This is its single most important characteristic.

XRef

The common process framework defines basic software engineering activities. It is described in Chapter 2.

As we noted earlier in this chapter, object-oriented software engineering applies a process model that encourages iterative development. That is, OO software evolves through a number of cycles. The common process framework that is used to manage an OO project must be evolutionary in nature.

Ed Berard [BER93] and Grady Booch [BOO91] among others suggest the use of a "recursive/parallel model" for object-oriented software development. In essence the recursive/parallel model works in the following way:

- Do enough analysis to isolate major problem classes and connections.
- Do a little design to determine whether the classes and connections can be implemented in a practical way.

How do we apply a recursive/parallel model for OO software engineering?

- Extract reusable objects from a library to build a rough prototype.
- Conduct some tests to uncover errors in the prototype.
- Get customer feedback on the prototype.
- Modify the analysis model based on what you've learned from the prototype, from doing design, and from customer feedback.
- Refine the design to accommodate your changes.
- Code special objects (that are not available from the library).
- Assemble a new prototype using objects from the library and the new objects you've created.
- Conduct some tests to uncover errors in the prototype.
- Get customer feedback on the prototype.

This approach continues until the prototype evolves into a production application.

The recursive/parallel model is quite similar to the spiral or evolutionary paradigm. Progress occurs iteratively. What makes the recursive/parallel model different is (1) the recognition that analysis and design modeling for OO systems cannot be accomplished at an even level of abstraction and (2) analysis and design can be applied to independent system components concurrently. Berard [BER93] describes the model in the following manner:

- Systematically decompose the problem into highly independent components.
- Reapply the decomposition process to each of the independent components to decompose each further (the recursive part).
- Conduct this reapplication of decomposition concurrently on all components (the parallel part).
- Continue this process until completion criteria are attained.

It's important to note that this decomposition process is discontinued if the analyst/designer recognizes that the component or subcomponent required is available in a reuse library.

In many ways, the architecture of an OO system makes it easier to initiate work in parallel. However, be certain that each parallel task is defined so that progress can be assessed.

To control the recursive/parallel process framework, the project manager must recognize that progress is planned and measured incrementally. That is, project tasks and the project schedule are tied to each of the "highly independent components," and progress is measured for each of these components individually.

Each iteration of the recursive/parallel process requires planning, engineering (analysis, design, class extraction, prototyping, and testing), and evaluation activities (Figure 20.11). During planning, activities associated with each of the independent program components are planned and scheduled. (Note: With each iteration, the schedule is adjusted to accommodate changes associated with the preceding iteration.) During early stages of engineering, analysis and design occur iteratively. The

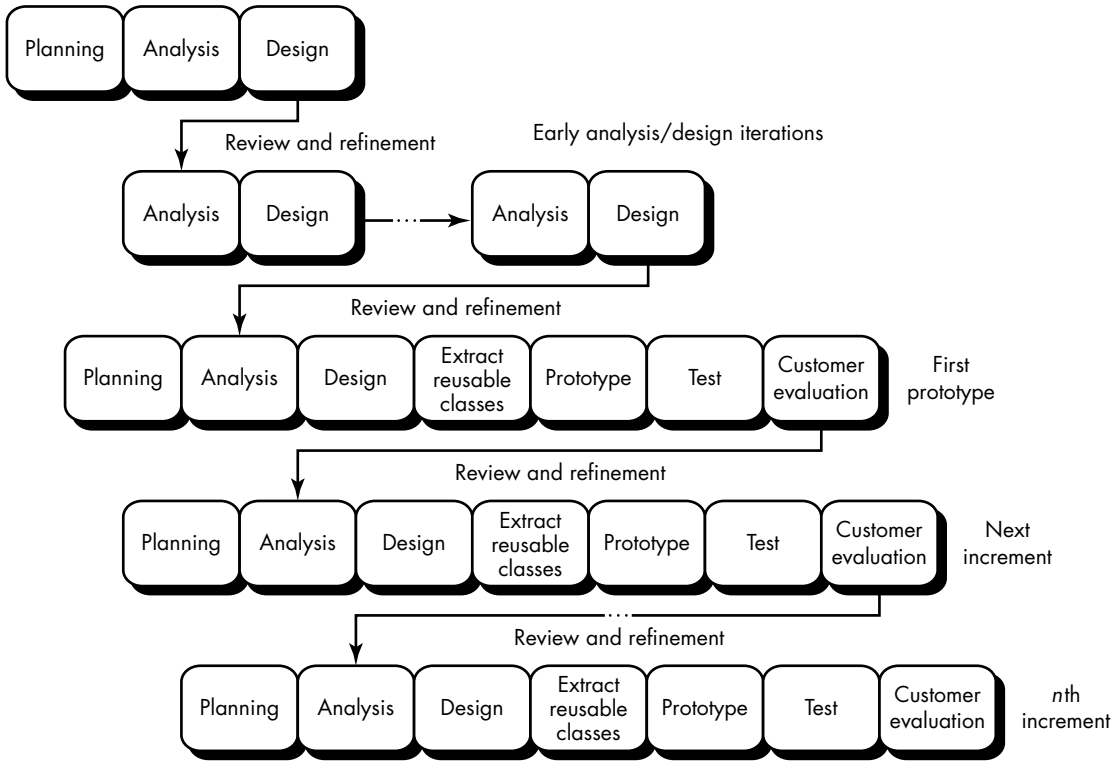


FIGURE 20.11 Typical process sequence for an OO project

intent is to isolate all important elements of the OO analysis and design models. As engineering work proceeds, incremental versions of the software are produced. During evaluation, reviews, customer evaluation, and testing are performed for each increment, with feedback affecting the next planning activity and subsequent increment.

20.4.2 OO Project Metrics and Estimation

Conventional software project estimation techniques require estimates of lines-of-code (LOC) or function points (FP) as the primary driver for estimation. Because an overriding goal for OO projects should be reuse, LOC estimates make little sense. FP estimates can be used effectively because the information domain counts that are required are readily obtainable from the problem statement. FP analysis may provide value for estimating OO projects, but the FP measure does not provide enough granularity for the schedule and effort adjustments that are required as we iterate through the recursive/parallel paradigm.

These metrics can be used to supplement the FP metric. They provide a way to "size" an OO project.

Lorenz and Kidd [LOR94] suggest the following set of project metrics:[6]

Number of scenario scripts. A *scenario script* (analogous to use-cases discussed in Chapter 11) is a detailed sequence of steps that describe the interaction between the user and the application. Each script is organized into triplets of the form

{**initiator,** *action,* **participant**}

where **initiator** is the object that requests some service (that initiates a message); *action* is the result of the request; and **participant** is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.

XRef

A detailed discussion of OO metrics is presented in Chapter 24.

Number of key classes. *Key classes* are the "highly independent components" [LOR94] that are defined early in OOA. Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Number of support classes. *Support classes* are required to implement the system but are not immediately related to the problem domain. Examples might be GUI classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout the recursive/parallel process.

The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.

Average number of support classes per key class. In general, key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be much simplified. Lorenz and Kidd suggest that applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.

Number of subsystems. A *subsystem* is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff.

---

6 Technical metrics for OO systems are discussed in detail in Chapter 24.

20.4.3 An OO Estimating and Scheduling Approach

Software project estimation remains more art than science. However, this in no way precludes the use of a systematic approach. To develop reasonable estimates it is essential to develop multiple data points. That is, estimates should be derived using a number of different techniques. Effort and duration estimates used for conventional software development are applicable to the OO world, but the historical database for OO projects is relatively small for many organizations. Therefore, it is worthwhile to supplement conventional software cost estimation with an approach that has been designed explicitly for OO software. Lorenz and Kidd [LOR94] suggest the following approach:

1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using OOA (Chapter 21), develop scenario scripts (use-cases) and determine a count. Recognize that the number of scenario scripts may change as the project progresses.
3. Using OOA, determine the number of key classes.
4. Categorize the type of interface for the application and develop a multiplier for support classes:

Interface type	Multiplier
No GUI	2.0
Text-based user interface	2.25
GUI	2.5
Complex GUI	3.0

   Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.
5. Multiply the total number of classes (key + support) by the average number of work-units per class. Lorenz and Kidd suggest 15 to 20 person-days per class.
6. Cross check the class-based estimate by multiplying the average number of work-units per scenario script.

XRef

A number of software project estimation techniques are considered in detail in Chapter 5.

Scheduling for object-oriented projects is complicated by the iterative nature of the process framework. Lorenz and Kidd suggest a set of metrics that may assist during project scheduling:

> **Number of major iterations.** Thinking back to the spiral model (Chapter 2), a major iteration would correspond to one 360° traversal of the spiral. The recursive/parallel process model would spawn a number of mini-spirals (localized iterations) that occur as the major iteration progresses. Lorenz and

Kidd suggest that iterations of between 2.5 and 4 months in length are easiest to track and manage.

Number of completed contracts. A *contract* is "a group of related public responsibilities that are provided by subsystems and classes to their clients" [LOR94]. A contract is an excellent milestone and at least one contract should be associated with each project iteration. A project manager can use completed contracts as a good indicator of progress on an OO project.

20.4.4 Tracking Progress for an OO Project

Although the recursive/parallel process model is the best framework for an OO project, task parallelism makes project tracking difficult. The project manager can have difficulty establishing meaningful milestones for an OO project because a number of different things are happening at once. In general, the following major milestones can be considered "completed" when the criteria noted have been met.

Technical milestone: OO analysis completed

- All classes and the class hierarchy have been defined and reviewed.
- Class attributes and operations associated with a class have been defined and reviewed.
- Class relationships (Chapter 21) have been established and reviewed.
- A behavioral model (Chapter 21) has been created and reviewed.
- Reusable classes have been noted.

Technical milestone: OO design completed

- The set of subsystems (Chapter 22) has been defined and reviewed.
- Classes are allocated to subsystems and reviewed.
- Task allocation has been established and reviewed.
- Responsibilities and collaborations (Chapter 22) have been identified.
- Attributes and operations have been designed and reviewed.
- The messaging model has been created and reviewed.

Technical milestone: OO programming completed

- Each new class has been implemented in code from the design model.
- Extracted classes (from a reuse library) have been implemented.
- Prototype or increment has been built.

Technical milestone: OO testing

- The correctness and completeness of OO analysis and design models has been reviewed.

- A class-responsibility-collaboration network (Chapter 23) has been developed and reviewed.
- Test cases are designed and class-level tests (Chapter 23) have been conducted for each class.
- Test cases are designed and cluster testing (Chapter 23) is completed and the classes are integrated.
- System level tests have been completed.

Recalling the recursive/parallel process model discussed earlier in this chapter, it is important to note that each of these milestones may be revisited as different increments are delivered to the customer.

20.5 SUMMARY

Object-oriented technologies reflect a natural view of the world. Objects are categorized into classes and class hierarchies. Each class contains a set of attributes that describe it and a set of operations that define its behavior. Objects model almost any identifiable aspect of the problem domain. External entities, things, occurrences, roles, organizational units, places, and structures can all be represented as objects. As important, objects (and the classes from which they are derived) encapsulate both data and process. Processing operations are part of the object and are initiated by passing the object a message. A class definition, once defined, forms the basis for reusability at the modeling, design, and implementation levels. New objects can be instantiated from a class.

Three important concepts differentiate the OO approach from conventional software engineering. Encapsulation packages data and the operations that manipulate the data into a single named object. Inheritance enables the attributes and operations of a class to be inherited by all subclasses and the objects that are instantiated from them. Polymorphism enables a number of different operations to have the same name, reducing the number of lines of code required to implement a system and facilitating changes when they are made.

Object-oriented products and systems are engineered using an evolutionary model, sometimes called a recursive/parallel model. OO software evolves iteratively and must be managed with the recognition that the final product will be developed over a series of increments.

REFERENCES

[BER93] Berard, E.V., *Essays on Object-Oriented Software Engineering,* Addison-Wesley, 1993.

[BOO86] Booch, G., "Object-Oriented Development," *IEEE Trans. Software Engineering,* vol. SE-12, no. 2, February 1986, pp. 211ff.

[BOO91] Booch, G., *Object-Oriented Design,* Benjamin Cummings, 1991.
[BUD96] Budd, T., *An Introduction to Object-Oriented Programming,* 2nd ed., Addison-Wesley, 1996.
[CAS89] Cashman, M., "Object Oriented Domain Analysis," *ACM Software Engineering Notes,* vol. 14, no. 6, October 1989, p. 67.
[CHA93] de Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development,* Addison-Wesley, 1993.
[COA91] Coad, P. and E. Yourdon, *Object-Oriented Analysis,* 2nd ed., Prentice-Hall, 1991.
[COX86] Cox, B.J., *Object-Oriented Programming,* Addison-Wesley, 1986.
[EVB89] *Object-Oriented Requirements Analysis* (course notebook), EVB Software Engineering, 1989.
[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.
[LOR94] Lorenz, M. and J. Kidd, *Object-Oriented Software Metrics,* Prentice-Hall, 1994.
[STR88] Stroustrup, B., "What Is Object-Oriented Programming?" *IEEE Software,* vol. 5, no. 3, May 1988, pp. 10–20.
[TAY90] Taylor, D.A., *Object-Oriented Technology: A Manager's Guide,* Addison-Wesley, 1990.

PROBLEMS AND POINTS TO PONDER

20.1. Object-oriented software engineering is rapidly displacing conventional software development approaches. Yet, like all technologies, OO has flaws. Using the Internet and hard-copy sources from your library, write a brief paper summarizing what critics have to say about OO and why they believe care must be taken when applying the OO paradigm.

20.2. In this chapter we did not consider the case in which a new object requires an attribute or operation that is not contained in the class from which it inherited all other attributes and operations. How do you think this is handled?

20.3. Do some research and find the real answer to Problem 20.2.

20.4. Using your own words and a few examples, define the terms *class, encapsulation, inheritance,* and *polymorphism.*

20.5. Review the objects defined for the *SafeHome* system. Are there other objects that you feel should be defined as modeling begins?

20.6. Consider a typical graphical user interface. Define a set of classes (and subclasses) for the interface entities that typically appear in the GUI. Be sure to define appropriate attributes and operations.

20.7. Provide an example of a composite object.

20.8. You have been assigned the job of engineering new word-processing software. A class named **document** is identified. Define the attributes and operations that are relevant for **document.**

20.9. Research two different OO programming languages and show how messages are implemented in the language syntax. Provide a few examples for each language.

20.10. Provide a concrete example of class hierarchy restructuring as described in the discussion of Figure 20.8.

20.11. Provide a concrete example of multiple inheritance. Research one or more papers on this subject and provide the pro and con arguments for multiple inheritance.

20.12. Develop a statement of scope for a system requested by your instructor. Use the grammatical parse to isolate candidate classes, attributes, and operations for the system. Apply the selection criteria discussed in Section 20.3.1 to determine whether the class should be used in the analysis model.

20.13. In your own words, describe why the recursive/parallel process model is appropriate for OO systems.

20.14. Provide three or four specific examples of the key class and support class described in Section 20.4.2.

FURTHER READINGS AND INFORMATION SOURCES

The explosion of interest in object-technologies has resulted in the publication of literally hundreds of books during the past 15 years. Taylor's abbreviated treatment [TAY90] remains a classic introduction to the subject. In addition, books by Ambler (*The Object Primer: The Application Developer's Guide to Object-Orientation,* SIGS Books, 1998), Gossain and Graham (*Object Modeling and Design Strategies,* SIGS Books, 1998), Bahar (*Object Technology Made Simple,* Simple Software Publishing, 1996), and Singer (*Object Technology Strategies and Tactics,* Cambridge University Press, 1996) are worthwhile introductions to object-oriented concepts and methods.

Zamir (*Handbook of Object Technology,* CRC Press, 1998) has edited a voluminous treatment that covers every aspect of object technologies. Fayad and Laitnen (*Transition to Object-Oriented Software Development,* Wiley, 1998) use case studies to identify technical, management, and cultural challenges that must be overcome when an organization makes the transition to object technologies. Gardner et al. (*Cognitive Patterns: Problem-Solving Frameworks for Object Technology,* Cambridge University Press, 1998) provide the reader with a basic introduction to problem-solving concepts and terminology associated with cognitive patterns and cognitive modeling as they are applied to OO systems.

The unique nature of the OO paradigm poses special challenges to project managers. Books by Cockburn (*Surviving Object-Oriented Projects: A Manager's Guide,* Addison-Wesley, 1998), Booch (*Object Solutions: Managing the Object-Oriented Project,* Addison-Wesley, 1995), Goldberg and Rubin (*Succeeding with Objects: Decision Frameworks for Project Management,* Addison-Wesley, 1995), and Meyer (*Object-Success: A Manager's Guide to Object-Orientation,* Prentice-Hall, 1995) consider strategies for planning, tracking, and controlling OO projects.

Eeles and Sims *(Building Business Objects,* Wiley, 1998), Carmichael (*Developing Business Objects,* SIGS Books, 1998), Fingar (*The Blueprint for Business Objects,* Cambridge University Press, 1996), and Taylor (*Business Engineering with Object Technology,* Wiley, 1995) address object technology as it is applied in a business context. Their books address methods for expressing business concepts and requirements directly as objects and object-oriented applications.

A wide variety of information sources on object technologies and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to OO can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/OO-concepts.mhtml

CHAPTER

21 OBJECT-ORIENTED ANALYSIS

KEY CONCEPTS

When a new product or system is to be built, how do we characterize it in a way that is amenable to object-oriented software engineering? Are there special questions that we need to ask the customer? What are the relevant objects? How do they relate to one another? How do objects behave in the context of the system? How do we specify or model a problem so that we can create an effective design?

Each of these questions is answered within the context of *object-oriented analysis* (OOA)—the first technical activity that is performed as part of OO software engineering. Instead of examining a problem using the classic information flow model, OOA introduces a number of new concepts. Coad and Yourdon [COA91] consider this issue when they write:

> OOA—object-oriented analysis—is based upon concepts that we first learned in kindergarten: objects and attributes, classes and members, wholes and parts. Why it has taken us so long to apply these concepts to the analysis and specification of information systems is anyone's guess . . .

OOA is grounded in a set of basic principles that were introduced in Chapter 11. In order to build an analysis model, five basic principles were applied: (1) the information domain is modeled; (2) function is described; (3) behavior is

QUICK LOOK

What is it? Before you can build an object-oriented system, you have to define the classes (objects) that represent the problem to be solved, the manner in which the classes relate to and interact with one another, the inner workings (attributes and operations) of objects, and the communication mechanisms (messages) that allow them to work together. All of these things are accomplished during object-oriented analysis (OOA).

Who does it? The definition of an object-oriented analysis model encompasses a description of the static and dynamic characteristics of classes that describe a system or product. This activity is performed by a software engineer.

Why is it important? You can't build software (object-oriented or otherwise) until you have a reasonable understanding of the system or product. OOA provides you with a concrete way to represent your understanding of requirements and then test that understanding against the customer's perception of the system to be built.

What are the steps? OOA begins with a description of use-cases—a scenario-based description of how actors (people, machines, other systems) interact with the product to be built. Class-responsibility-collaborator (CRC) modeling translates the information contained in use-cases into a representation of classes and their collaborations with other classes. The static and dynamic characteristics of classes are then modeled using ▶

QUICK LOOK

a unified modeling language (or some other method).

What is the work product? An object-oriented analysis model is created. The OO analysis model is composed of graphical or language-based representations that define class attributes, relationships, and behaviors, as well as interclass communication and a depiction of class behavior over time.

How do I ensure that I've done it right? At each stage, the elements of the object-oriented analysis model are reviewed for clarity, correctness, completeness, and consistency with customer requirements and with one another.

represented; (4) data, functional, and behavioral models are partitioned to expose greater detail; and (5) early models represent the essence of the problem while later models provide implementation details. These principles form the foundation for the approach to OOA presented in this chapter.

The intent of OOA is to define all classes that are relevant to the problem to be solved—the operations and attributes associated with them, the relationships between them, and behavior they exhibit. To accomplish this, a number of tasks must occur:

Quote:

"A problem well-stated is a problem half-solved."

Charles Kettering

1. Basic user requirements must be communicated between the customer and the software engineer.
2. Classes must be identified (i.e., attributes and methods are defined).
3. A class hierarchy must be specified.
4. Object-to-object relationships (object connections) should be represented.
5. Object behavior must be modeled.
6. Tasks 1 through 5 are reapplied iteratively until the model is complete.

It is important to note that there is no universal agreement on the "concepts" that serve as a foundation for OOA. But a limited number of key ideas appear repeatedly, and it is these that we will consider in this chapter.

21.1 OBJECT-ORIENTED ANALYSIS

The objective of object-oriented analysis is to develop a model that describes computer software as it works to satisfy a set of customer-defined requirements. OOA, like the conventional analysis methods described in Chapter 12, builds a multipart analysis model to satisfy this objective. The analysis model depicts information, function, and behavior within the context of the elements of the object model described in Chapter 20.

21.1.1 Conventional vs. OO Approaches

Is object-oriented analysis really different from the structured analysis approach that was presented in Chapter 12? Fichman and Kemerer [FIC92] address the question head-on:

> We conclude that the object-oriented analysis approach . . . represents a radical change over process oriented methodologies such as structured analysis, but only an incremental change over data oriented methodologies such as information engineering. Process-oriented methodologies focus attention away from the inherent properties of objects during the modeling process and lead to a model of the problem domain that is orthogonal to the three essential principles of object-orientation: encapsulation, classification of objects, and inheritance.

Stated simply, structured analysis (SA) takes a distinct input-process-output view of requirements. Data are considered separately from the processes that transform the data. System behavior, although important, tends to play a secondary role in structured analysis. The structured analysis approach makes heavy use of functional decomposition (partitioning of the data flow diagram, Chapter 12).

Fichman and Kemerer [FIC92] suggest 11 "modeling dimensions" that may be used to compare various conventional and object-oriented analysis methods:

What criteria can be used to compare conventional and OOA methods?

1. Identification/classification of entities[1]
2. General-to-specific and whole-to-part entity relationships
3. Other entity relationships
4. Description of attributes of entities
5. Large-scale model partitioning
6. States and transitions between states
7. Detailed specification for functions
8. Top-down decomposition
9. End-to-end processing sequences
10. Identification of exclusive services
11. Entity communication (via messages or events)

Because many variations exist for structured analysis and dozens of OOA methods (see Section 21.1.2) have been proposed over the years, it is difficult to develop a generalized comparison between the two methods. It can be stated, however, that modeling dimensions 8 and 9 are always present with SA and never present when OOA is used.

21.1.2 The OOA Landscape

The popularity of object technologies spawned dozens of OOA methods during the late 1980s and into the 1990s.[2] Each of these introduced a process for the analysis

---

1 In this context, *entity* refers to either a data object (in the structured analysis sense) or an object (in the OOA sense).

2 A detailed discussion of these methods and their differences is beyond the scope of this book. In addition, the industry is moving toward a unified method of analysis modeling, making a detailed discussion of older methods useful for historical purposes only. The interested reader should refer to Berard [BER99] and Graham [GRA94] for detailed comparisons.

of a product or system, a set of diagrams that evolved out of the process, and a notation that enabled the software engineer to create the analysis model in a consistent manner. Among the most widely used were[3]

The Booch method. The Booch method [BOO94] encompasses both a "micro development process" and a "macro development process." The micro level defines a set of analysis tasks that are reapplied for each step in the macro process. Hence, an evolutionary approach is maintained. Booch's OOA micro development process identifies classes and objects and the semantics of classes and objects and defines relationships among classes and objects and conducts a series of refinements to elaborate the analysis model.

Quote:

"The central activity of working with objects is not so much a matter of programming as it is representation."

David Taylor

The Rumbaugh method. Rumbaugh [RUM91] and his colleagues developed the *object modeling technique* (OMT) for analysis, system design, and object-level design. The analysis activity creates three models: the object model (a representation of objects, classes, hierarchies, and relationships), the dynamic model (a representation of object and system behavior), and the functional model (a high-level DFD-like representation of information flow through the system).

The Jacobson method. Also called OOSE (object-oriented software engineering), the Jacobson method [JAC92] is a simplified version of the proprietary objectory method, also developed by Jacobson. This method is differentiated from others by heavy emphasis on the use-case—a description or scenario that depicts how the user interacts with the product or system.

The Coad and Yourdon method. The Coad and Yourdon method [COA91] is often viewed as one of the easiest OOA methods to learn. Modeling notation is relatively simple and guidelines for developing the analysis model are straightforward. A brief outline of Coad and Yourdon's OOA process follows:

- Identify objects using "what to look for" criteria.
- Define a generalization/specification structure.
- Define a whole/part structure.
- Identify subjects (representations of subsystem components).
- Define attributes.
- Define services.

The Wirfs-Brock method. Wirfs-Brock, Wilkerson, and Weiner [WIR90] do not make a clear distinction between analysis and design tasks. Rather a continuous process that begins with the assessment of a customer specification and ends with design is proposed. A brief outline of Wirfs-Brock et al.'s analysis-related tasks follows:

3 In general, OOA methods are identified using the name(s) of the developer of the method, even if the method has been given a unique name or acronym.

- Evaluate the customer specification.
- Extract candidate classes from the specification via grammatical parsing.
- Group classes in an attempt to identify superclasses.
- Define responsibilities for each class.
- Assign responsibilities to each class.
- Identify relationships between classes.
- Define collaboration between classes based on responsibilities.
- Build hierarchical representations of classes.
- Construct a collaboration graph for the system.

Although the terminology and process steps for each of these OOA methods differ, the overall OOA processes are really quite similar. To perform object-oriented analysis, a software engineer should perform the following generic steps:

A set of generic steps are applied during OOA, regardless of the analysis method that is chosen.

1. Elicit customer requirements for the system.
2. Identify scenarios or use-cases.
3. Select classes and objects using basic requirements as a guide.
4. Identify attributes and operations for each system object.
5. Define structures and hierarchies that organize classes.
6. Build an object-relationship model.
7. Build an object-behavior model.
8. Review the OO analysis model against use-cases or scenarios.

These generic steps are considered in greater detail in Sections 21.3 and 21.4.

21.1.3 A Unified Approach to OOA

"UML has unified some of the existing OO notations, thus creating a single point of reference for many important concepts."

Peter Hruschka

Over the past decade, Grady Booch, James Rumbaugh, and Ivar Jacobson have collaborated to combine the best features of their individual object-oriented analysis and design methods into a unified method. The result, called the *Unified Modeling Language* (UML), has become widely used throughout the industry.[4]

UML allows a software engineer to express an analysis model using a modeling notation that is governed by a set of syntactic, semantic, and pragmatic rules. Eriksson and Penker [ERI98] explain these rules in the following way:

> The syntax tells us how the symbols should look and how the symbols are combined. The syntax is compared to words in natural language; it is important to know how to spell them correctly and how to put different words together to form a sentence. The semantic rules tell us what each symbol means and how it should be interpreted by itself and in the context of other symbols; they are compared to the meanings of words in a natural language.

4 Booch, Rumbaugh, and Jacobson have written a set of three definitive books on UML. The interested reader should see [BOO99], [RUM99], and [JAC99].

The pragmatic rules define the intentions of the symbols through which the purpose of the model is achieved and becomes understandable for others. This corresponds in natural language to the rules for constructing sentences that are clear and understandable.

In UML, a system is represented using five different "views" that describe the system from distinctly different perspectives. Each view is defined by a set of diagrams. The following views [ALH98] are present in UML:

Like all analysis approaches, requirements elicitation is key. Be certain that you get the user model view right. The rest will follow.

User model view. This view represents the system (product) from the user's (called *actors* in UML) perspective. The use-case is the modeling approach of choice for the user model view. This important analysis representation describes a usage scenario from the end-user's perspective and has been discussed in detail in Chapter 11.[5]

Structural model view. Data and functionality are viewed from inside the system. That is, static structure (classes, objects, and relationships) is modeled.

Behavioral model view. This part of the analysis model represents the dynamic or behavioral aspects of the system. It also depicts the interactions or collaborations between various structural elements described in the user model and structural model views.

Implementation model view. The structural and behavioral aspects of the system are represented as they are to be built.

Environment model view. The structural and behavioral aspects of the environment in which the system is to be implemented are represented.

An extensive tutorial and listing of UML resources including tools, papers, and examples can be found at **mini.net/cetus/oo_uml.html**

In general, UML analysis modeling focuses on the user model and structural model views of the system. UML design modeling (considered in Chapter 22) addresses the behavioral model, implementation model, and environmental model views.

21.2 DOMAIN ANALYSIS

The objective of domain analysis is to define a set of classes (objects) that are encountered throughout an application domain. These can then be reused in many applications.

Analysis for object-oriented systems can occur at many different levels of abstraction. At the business or enterprise level, the techniques associated with OOA can be coupled with a business process engineering approach (Chapter 10) in an effort to define classes, objects, relationships, and behaviors that model the entire business. At the business area level, an object model that describes the workings of a particular business area (or a category of products or systems) can be defined. At an application level, the object model focuses on specific customer requirements as those requirements affect an application to be built.

OOA at the highest level of abstraction (the enterprise level) is beyond the scope of this book. Interested readers should see [EEL98], [CAR98], [FIN96], [TAY95], [MAT94],

5 If you have not already done so, please read Section 11.2.4 for a detailed discussion of use-cases.

and [SUL94] for detailed discussions of enterprise-level modeling. OOA at the lowest level of abstraction falls within the general purview of object-oriented software engineering and is the focus of all other sections of this chapter. In this section, we conducted OOA at a middle level of abstraction. This activity, called *domain analysis,* is performed when an organization wants to create a library of reusable classes (components) that will be broadly applicable to an entire category of applications.

21.2.1 Reuse and Domain Analysis

Object-technologies are leveraged through reuse. Consider a simple example. The analysis of requirements for a new application indicates that 100 classes are needed. Two teams are assigned to build the application. Each will design and construct a final product. Each team is populated by people with the same skill levels and experience.

Team A does not have access to a class library, and therefore, it must develop all 100 classes from scratch. Team B uses a robust class library and finds that 55 classes already exist. It is highly likely that

Other benefits derived from reuse are consistency and familiarity. Patterns within the software will become more consistent, leading to better maintainability. Be certain to establish a set of reuse "design rules" so that these benefits are achieved.

1. Team B will finish the project much sooner than Team A.
2. The cost of Team B's product will be significantly lower than the cost of Team A's product.
3. The product produced by Team B will have fewer delivered defects than Team A's product.

Although the margin by which Team B's work would exceed Team A's accomplishments is open to debate, few would argue that reuse provides Team B with a substantial advantage.

But where did the "robust class library" come from? How were the entries in the library determined to be appropriate for use in new applications? To answer these questions, the organization that created and maintained the library had to apply domain analysis.

21.2.2 The Domain Analysis Process

Firesmith [FIR93] describes software domain analysis in the following way:

> Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain . . . [Object-oriented domain analysis is] the identification, analysis, and specification of common, reusable capabilities within a specific application domain, in terms of common objects, classes, subassemblies, and frameworks . . .

XRef

Reuse is the cornerstone of component-based software engineering, a topic discussed in Chapter 27.

The "specific application domain" can range from avionics to banking, from multimedia video games to applications within an MRI device. The goal of domain analysis is straightforward: to find or create those classes that are broadly applicable, so that they may be reused.

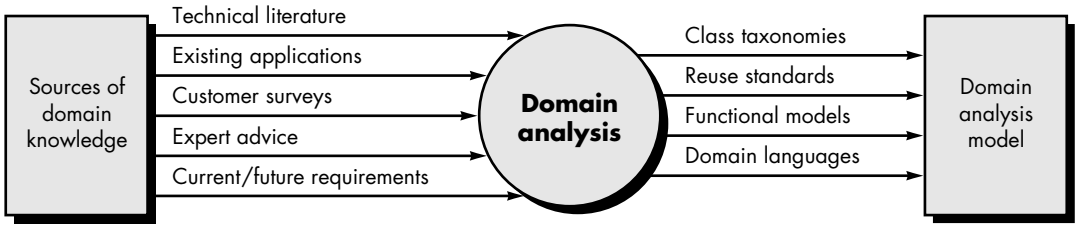


FIGURE 21.1 Input and output for domain analysis

Using terminology that was introduced earlier in this book, domain analysis may be viewed as an umbrella activity for the software process. By this we mean that domain analysis is an ongoing software engineering activity that is not connected to any one software project. In a way, the role of a domain analyst is similar to the role of a master toolsmith in a heavy manufacturing environment. The job of the toolsmith is to design and build tools that may be used by many people doing similar but not necessarily the same jobs. The role of the domain analyst is to design and build reusable components that may be used by many people working on similar but not necessarily the same applications.

Quote:

"If an organization is to make a major investment in software reuse, it needs to know what components to consider in the development of such a model."

David Rine

Figure 21.1 [ARA89] illustrates key inputs and outputs for the domain analysis process. Sources of domain knowledge are surveyed in an attempt to identify objects that can be reused across the domain. In essence domain analysis is quite similar to knowledge engineering. The knowledge engineer investigates a specific area of interest in an attempt to extract key facts that may be of use in creating an expert system or artificial neural network. During domain analysis, *object* (and class) *extraction* occurs.

The domain analysis process can be characterized by a series of activities that begin with the identification of the domain to be investigated and end with a specification of the objects and classes that characterize the domain. Berard [BER93] suggests the following activities:

Define the domain to be investigated. To accomplish this, the analyst must first isolate the business area, system type, or product category of interest. Next, both OO and non-OO "items" must be extracted. OO items include specifications, designs, and code for existing OO application classes; support classes (e.g., GUI classes or database access classes); commercial off-the-shelf (COTS) component libraries that are relevant to the domain; and test cases. Non-OO items encompass policies, procedures, plans, standards, and guidelines; parts of existing non-OO applications (including specification, design, and test information); metrics; and COTS non-OO software.

XRef

A complete domain analysis strategy must consider architecture as well as components. A detailed discussion of software architecture is presented in Chapter 14.

Categorize the items extracted from the domain. The items are organized into categories and the general defining characteristics of the category are defined. A classification scheme for the categories is proposed and nam-

ing conventions for each item are defined. When appropriate, classification hierarchies are established.

Collect a representative sample of applications in the domain. To accomplish this activity, the analyst must ensure that the application in question has items that fit into the categories that have already been defined. Berard [BER93] notes that during the early stages of use of object-technologies, a software organization will have few if any OO applications. Therefore, the domain analyst must "identify the conceptual (as opposed to physical) objects in each [non-OO] application."

Analyze each application in the sample. The following steps [BER93] are followed by the analyst:

- Identify candidate reusable objects.
- Indicate the reasons that the object has been identified for reuse.
- Define adaptations to the object that may also be reusable.
- Estimate the percentage of applications in the domain that might make reuse of the object.
- Identify the objects by name and use configuration management techniques (Chapter 9) to control them. In addition, once the objects have been defined, the analyst should estimate what percentage of a typical application could be constructed using the reusable objects.

Develop an analysis model for the objects. The analysis model will serve as the basis for design and construction of the domain objects.

WebRef

A worthwhile tutorial on domain analysis can be found at **www.sei.cmu.edu/str/descriptions/deda.html**

In addition to these steps, the domain analyst should also create a set of reuse guidelines and develop an example that illustrates how the domain objects could be used to create a new application.

Domain analysis is the first technical activity in a broader discipline that some call *domain engineering*. When a business, system, or product domain is defined to be business strategic in the long term, a continuing effort to create a robust reuse library can be undertaken. The goal is to be able to create software within the domain with a very high percentage of reusable components. Lower cost, higher quality, and improved time to market are the arguments in favor of a dedicated domain engineering effort.

21.3 GENERIC COMPONENTS OF THE OO ANALYSIS MODEL

The object-oriented analysis process conforms to the basic analysis concepts and principles discussed in Chapter 11. Although the terminology, notation, and

activities differ from conventional methods, OOA (at its kernel) addresses the same underlying objectives. Rumbaugh et al. [RUM91] discuss this when they state:

> Analysis . . . is concerned with devising a precise, concise, understandable, and correct model of the real world. . . . The purpose of object-oriented analysis is to model the real world so that it can be understood. To do this, you must examine requirements, analyze their implications, and restate them rigorously. You must abstract real-world features first, and defer small details until later.

To develop a "precise, concise, understandable, and correct model of the real world," a software engineer must select a notation that implements a set of generic components of an OO analysis model. Monarchi and Puhr [MON92] define a set of generic representational components that appear in all OO analysis models.[6] *Static components* are structural in nature and indicate characteristics that hold throughout the operational life of an application. These characteristics distinguish one object from other objects. *Dynamic components* focus on control and are sensitive to timing and event processing. They define how one object interacts with other objects over time. The following components are identified [MON92]:

What are the key components of an OOA model?

Static view of semantic classes. A taxonomy of typical classes was identified in Chapter 20. Requirements are assessed and classes are extracted (and represented) as part of the analysis model. These classes persist throughout the life of the application and are derived based on the semantics of the customer requirements.

Static view of attributes. Every class must be explicitly described. The attributes associated with the class provide a description of the class, as well as a first indication of the operations that are relevant to the class.

Static view of relationships. Objects are "connected" to one another in a variety of ways. The analysis model must represent these relationships so that operations (that affect these connections) can be identified and the design of a messaging approach can be accomplished.

KEY POINT

Static components do not change as the application is executed. Dynamic components are influenced by timing and events.

Static view of behaviors. The relationships just noted define a set of behaviors that accommodate the usage scenario (use-cases) of the system. These behaviors are implemented by defining a sequence of operations that achieve them.

Dynamic view of communication. Objects must communicate with one another and do so based on a series of events that cause transition from one state of a system to another

Dynamic view of control and time. The nature and timing of events that cause transitions among states must be described.

---

6 The authors [MON92] also provide an analysis of 23 early OOA methods and indicate how they address these components.

De Champeaux, Lea, and Faure [CHA93] define a slightly different view of OOA representations. Static and dynamic components are identified for object internals and for interobject representations. A dynamic view of object internals can be characterized as an *object life history;* that is, the states of the object change over time as various operations are performed on its attributes.

21.4 THE OOA PROCESS

The OOA process does not begin with a concern for objects. Rather, it begins with an understanding of the manner in which the system will be used—by people, if the system is human-interactive; by machines, if the system is involved in process control; or by other programs, if the system coordinates and controls applications. Once the scenario of usage has been defined, the modeling of the software begins.

The sections that follow define a series of techniques that may be used to gather basic customer requirements and then define an analysis model for an object-oriented system.

21.4.1 Use-Cases

As we noted in Chapter 11, use-cases model the system from the end-user's point of view. Created during requirements elicitation, use-cases should achieve the following objectives:

XRef

Use-cases are an excellent requirements elicitation tool, regardless of the analysis method that is used. See Chapter 11 for additional information.

- To define the functional and operational requirements of the system (product) by defining a scenario of usage that is agreed upon by the end-user and the software engineering team.
- To provide a clear and unambiguous description of how the end-user and the system interact with one another.
- To provide a basis for validation testing.

During OOA, use-cases serve as the basis for the first element of the analysis model.

Using UML notation, a diagrammatic representation of a use-case, called a *use-case diagram,* can be created. Like many elements of the analysis model, the use-case diagram can be represented at many levels of abstraction. The use-case diagram contains actors and use-cases. *Actors* are entities that interact with the system. They can be human users or other machines or systems that have defined interfaces to the software.

To illustrate the development of a use-case diagram, we consider the use-cases for the *SafeHome* security system described in Section 11.2.4. Three actors were identified: the **homeowner, sensors,** and the **monitoring and response subsystem**. For the purpose of this example, only the homeowner is considered.

Figure 21.2A depicts a high-level use-case diagram for the **homeowner**. Referring to Figure 21.2A, two use-cases are identified (represented by ovals). Each of the high-level use-cases may be elaborated with lower-level use-case diagrams. For

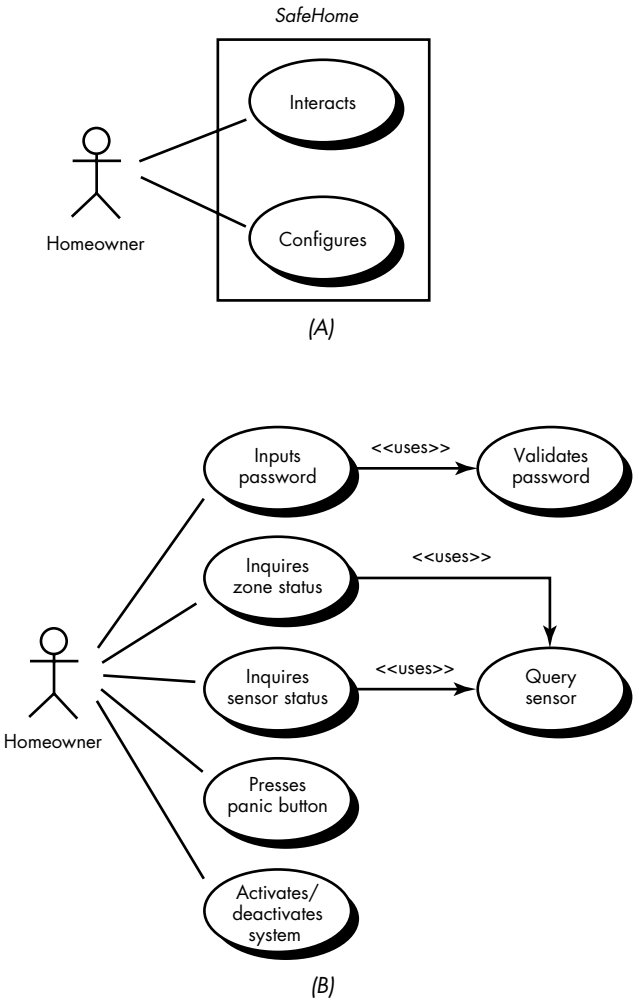


FIGURE 21.2
(A) High-level use-case diagram, (B) elaborated use-case diagram

example, Figure 21.2B represents a use-case diagram that elaborates the *interacts* function. A complete set of use-case diagrams is created for all actors. A detailed discussion of use-case modeling using UML is best left to books (e.g., [ERI98], [ALH98]) dedicated to this OOA method.

21.4.2 Class-Responsibility-Collaborator Modeling

Once basic usage scenarios have been developed for the system, it is time to identify candidate classes and indicate their responsibilities and collaborations. *Class-responsibility-collaborator* (CRC) modeling [WIR90] provides a simple means for identifying and organizing the classes that are relevant to system or product requirements. Ambler [AMB95] describes CRC modeling in the following way:

A CRC model is really a collection of standard index cards that represent classes. The cards are divided into three sections. Along the top of the card you write the name of the class. In the body of the card you list the class responsibilities on the left and the collaborators on the right.

In reality, the CRC model may make use of actual or virtual *index cards.* The intent is to develop an organized representation of classes. *Responsibilities* are the attributes and operations that are relevant for the class. Stated simply, a responsibility is "anything the class knows or does" [AMB95]. *Collaborators* are those classes that are required to provide a class with the information needed to complete a responsibility. In general, a collaboration implies either a request for information or a request for some action.

Classes

Basic guidelines for identifying classes and objects were presented in Chapter 20. To summarize, objects manifest themselves in a variety of forms (Section 20.3.1): external entities, things, occurrences, or events; roles; organizational units; places; or structures. One technique for identifying these in the context of a software problem is to perform a grammatical parse on the processing narrative for the system. All nouns become potential objects. However, not every potential object makes the cut. Six selection characteristics were defined:

? How do I determine whether a potential object is worthy of inclusion on a CRC index card?

1. **Retained information.** The potential object will be useful during analysis only if information about it must be remembered so that the system can function.
2. **Needed services.** The potential object must have a set of identifiable operations that can change the value of its attributes in some way.
3. **Multiple attributes.** During requirements analysis, the focus should be on "major" information; an object with a single attribute may, in fact, be useful during design but is probably better represented as an attribute of another object during the analysis activity.
4. **Common attributes.** A set of attributes can be defined for the potential object and these attributes apply to all occurrences of the object.
5. **Common operations.** A set of operations can be defined for the potential object and these operations apply to all occurrences of the object.
6. **Essential requirements.** External entities that appear in the problem space and produce or consume information that is essential to the operation of any solution for the system will almost always be defined as objects in the requirements model.

A potential object should satisfy all six of these selection characteristics if it is to be considered for inclusion in the CRC model.

Firesmith [FIR93] extends this taxonomy of class types by suggesting the following additions:

? Is there a way to categorize classes, and what characteristics help us do this?

Device classes model external entities such as sensors, motors, keyboards.

Property classes represent some important property of the problem environment (e.g., credit rating within the context of a mortgage loan application).

Interaction classes model interactions that occur among other objects (e.g., a purchase or a license).

In addition, objects and classes may be categorized by a set of characteristics:

Tangibility. Does the class represent a tangible thing (e.g., a keyboard or sensor) or does it represent more abstract information (e.g., a predicted outcome)?

Inclusiveness. Is the class *atomic* (i.e., it includes no other classes) or is it *aggregate* (it includes at least one nested object)?

Sequentiality. Is the class concurrent (i.e., it has its own thread of control) or sequential (it is controlled by outside resources)?

Persistence. Is the class *transient* (i.e., it is created and removed during program operation), *temporary* (it is created during program operation and removed once the program terminates), or *permanent* (it is stored in a database)?

Integrity. Is the class *corruptible* (i.e., it does not protect its resources from outside influence) or *guarded* (i.e., the class enforces controls on access to its resources)?

Using these class categories, the "index card" created as part of the CRC model might be extended to include the type of class and its characteristics (Figure 21.3).

Responsibilities

KEY POINT

The responsibilities of a class encompass both attributes and operations.

Basic guidelines for identifying responsibilities (attributes and operations) were also presented in Chapter 20. To summarize, attributes represent stable features of a class; that is, information about the class that must be retained to accomplish the objectives of the software specified by the customer. Attributes can often be extracted from the statement of scope or discerned from an understanding of the nature of the class. Operations can be extracted by performing a grammatical parse on the processing narrative for the system. All verbs become candidate operations. Each operation that is chosen for a class exhibits a behavior of the class.

Wirfs-Brock and her colleagues [WIR90] suggest five guidelines for allocating responsibilities to classes:

FIGURE 21.3
A CRC model index card

Class name:	
Class type: (e.g., device, property, role, event)	
Class characteristic: (e.g., tangible, atomic, concurrent)	
responsibilities:	collaborations:

How do I allocate responsibilities to a class?

1. **System intelligence should be evenly distributed.** Every application encompasses a certain degree of intelligence; that is, what the system knows and what it can do. This intelligence can be distributed across classes in a number of different ways. "Dumb" classes (those that have few responsibilities) can be modeled to act as servants to a few "smart" classes (those having many responsibilities. Although this approach makes the flow of control in a system straightforward, it has a few disadvantages: (1) It concentrates all intelligence within a few classes, making changes more difficult, and (2) it tends to require more classes, hence more development effort.

   Therefore, system intelligence should be evenly distributed across the classes in an application. Because each object knows about and does only a few things (that are generally well focused), the cohesiveness of the system is improved. In addition, side effects due to change tend to be dampened because system intelligence has been decoupled across many objects.

ADVICE

If a class has an extraordinarily long list of responsibilities, you should consider partitioning its definition into more than one class.

   To determine whether system intelligence is evenly distributed, the responsibilities noted on each CRC model index card should be evaluated to determine if any class has an extraordinarily long list of responsibilities. This indicates a concentration of intelligence. In addition, the responsibilities for each class should exhibit the same level of abstraction. For example, among the operations listed for an aggregate class called **checking account** a reviewer notes two responsibilities: *balance-the-account* and *check-off-*

cleared-checks. The first operation (responsibility) implies a complex mathematical and logical procedure. The second is a simple clerical activity. Since these two operations are not at the same level of abstraction, *check-off-cleared-checks* should be placed within the responsibilities of **check-entry**, a class that is encompassed by the aggregate class **checking account**.

2. **Each responsibility should be stated as generally as possible.** This guideline implies that general responsibilities (both attributes and operations) should reside high in the class hierarchy (because they are generic, they will apply to all subclasses). In addition, polymorphism (Chapter 20) should be used in an effort to define operations that generally apply to the superclass but are implemented differently in each of the subclasses.
3. **Information and the behavior related to it should reside within the same class.** This achieves the OO principle that we have called *encapsulation* (Chapter 20). Data and the processes that manipulate the data should be packaged as a cohesive unit.
4. **Information about one thing should be localized with a single class, not distributed across multiple classes.** A single class should take on the responsibility for storing and manipulating a specific type of information. This responsibility should not, in general, be shared across a number of classes. If information is distributed, software becomes more difficult to maintain and more challenging to test.
5. **Responsibilities should be shared among related classes, when appropriate.** There are many cases in which a variety of related objects must all exhibit the same behavior at the same time. As an example, consider a video game that must display the following objects: **player, player-body, player-arms, player-legs, player-head.** Each of these objects has its own attributes (e.g., **position, orientation, color, speed**) and all must be updated and displayed as the user manipulates a joy stick. The responsibilities *update* and *display* must therefore be shared by each of the objects noted. **Player** knows when something has changed and *update* is required. It collaborates with the other objects to achieve a new position or orientation, but each object controls its own display.

Collaborations

Classes fulfill their responsibilities in one of two ways: (1) A class can use its own operations to manipulate its own attributes, thereby fulfilling a particular responsibility, or (2) a class can collaborate with other classes.

Wirfs-Brock and her colleagues [WIR90] define collaborations in the following way:

> Collaborations represent requests from a client to a server in fulfillment of a client responsibility. A collaboration is the embodiment of the contract between the client and the server. . . . We say that an object collaborates with another object if, to fulfill a responsibility, it

needs to send the other object any messages. A single collaboration flows in one direction—representing a request from the client to the server. From the client's point of view, each of its collaborations are associated with a particular responsibility implemented by the server.

A server object collaborates with a client object in an effort to fulfill some responsibility. The collaboration involves the passing of messages.

Collaborations identify relationships between classes. When a set of classes all collaborate to achieve some requirement, they can be organized into a subsystem (a design issue).

Collaborations are identified by determining whether a class can fulfill each responsibility itself. If it cannot, then it needs to interact with another class. Hence, a collaboration.

As an example, consider the *SafeHome* application.[7] As part of the activation procedure (see the use-case for *activation* in Section 11.2.4), the **control panel** object must determine whether any sensors are open. A responsibility named *determine-sensor-status* is defined. If sensors are open **control panel** must set a **status** attribute to "not ready." Sensor information can be acquired from the **sensor** object. Therefore, the responsibility *determine-sensor-status* can be fulfilled only if **control panel** works in collaboration with **sensor**.

To help in the identification of collaborators, the analyst can examine three different generic relationships between classes [WIR90]: (1) the *is-part-of* relationship, (2) the *has-knowledge-of* relationship, and (3) the *depends-upon* relationship. By creating a class-relationship diagram (Section 21.4.4), the analyst develops the connections necessary to identify these relationships. Each of the three generic relations is considered briefly in the paragraphs that follow.

All classes that are part of an aggregate class are connected to the aggregate class via an is-part-of relationship. Consider the classes defined for the video game noted earlier, the class **player-body** is-part-of **player,** as are **player-arms, player-legs,** and **player-head.**

When one class must acquire information from another class, the has-knowledge-of relationship is established. The *determine-sensor-status* responsibility noted earlier is an example of a has-knowledge-of relationship.

The depends-upon relationship implies that two classes have a dependency that is not achieved by has-knowledge-of or is-part-of. For example, **player-head** must always be connected to **player-body** (unless the video game is particularly violent), yet each object could exist without direct knowledge of the other. An attribute of the **player-head** object called **center-position** is determined from the center position of **player-body**. This information is obtained via a third object, **player**, that acquires it from **player-body**. Hence, **player-head** depends-upon **player-body**.

In all cases, the collaborator class name is recorded on the CRC model index card next to the responsibility that has spawned the collaboration. Therefore, the index card contains a list of responsibilities and the corresponding collaborations that enable the responsibilities to be fulfilled (Figure 21.3).

---

7 See Section 20.3 for a delineation of classes for *SafeHome*.

When a complete CRC model has been developed, the representatives from the customer and software engineering organizations can review the model using the following approach [AMB95]:

? What is an effective approach for reviewing a CRC model?

1. All participants in the review (of the CRC model) are given a subset of the CRC model index cards. Cards that collaborate should be separated (i.e., no reviewer should have two cards that collaborate).
2. All use-case scenarios (and corresponding use-case diagrams) should be organized into categories.
3. The review leader reads the use-case deliberately. As the review leader comes to a named object, she passes a token to the person holding the corresponding class index card. For example, a use-case for *SafeHome* contains the following narrative:

   > The homeowner observes the *SafeHome* control panel to determine if the system is ready for input. If the system is not ready, the homeowner must physically close windows/doors so that the ready indicator is present. [A not-ready indicator implies that a sensor is open, i.e., that a door or window is open.]

   When the review leader comes to "control panel," in the use-case narrative, the token is passed to the person holding the **control panel** index card. The phrase "implies that a sensor is open" requires that the index card contains a responsibility that will validate this implication (the responsibility *determine-sensor-status* accomplishes this). Next to the responsibility on the index card is the collaborator **sensor**. The token is then passed to the **sensor** object.
4. When the token is passed, the holder of the class card is asked to describe the responsibilities noted on the card. The group determines whether one (or more) of the responsibilities satisfies the use-case requirement.
5. If the responsibilities and collaborations noted on the index cards cannot accommodate the use-case, modifications are made to the cards. This may include the definition of new classes (and corresponding CRC index cards) or the specification of new or revised responsibilities or collaborations on existing cards.

This modus operandi continues until the use-case is finished. When all use-cases (or use-case diagrams) have been reviewed, OOA continues.

21.4.3 Defining Structures and Hierarchies

Once classes and objects have been identified using the CRC model, the analyst begins to focus on the structure of the class model and the resultant hierarchies that arise as classes and subclasses emerge. Using UML notation, a variety of class diagrams can be created. *Generalization/specialization* class structures can be created for identified classes.

To illustrate, consider the **sensor** object defined for *SafeHome*, shown in Figure 21.4. Here, the generalization class, **sensor**, is refined into a set of specializations—

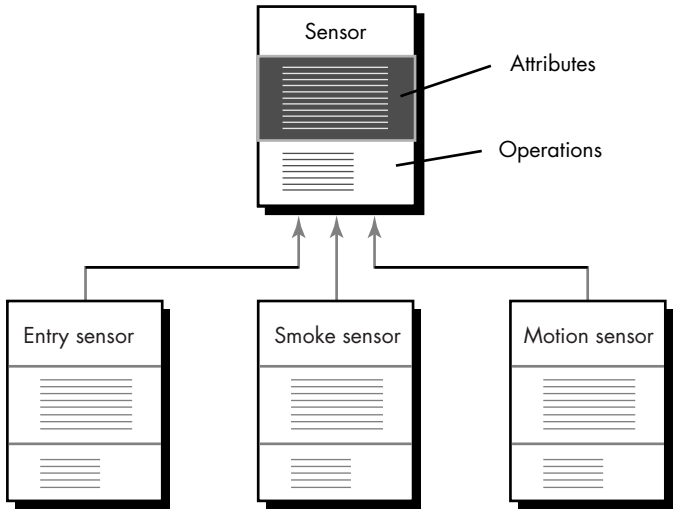


FIGURE 21.4
Class diagram for generalization/specialization

entry sensor, smoke sensor, and **motion sensor**. The attributes and operations noted for the **sensor** class are inherited by the specializations of the class. We have created a simple class hierarchy.

In other cases, an object represented in the initial model might actually be composed of a number of component parts that could themselves be defined as objects. These aggregate objects can be represented as a *composite aggregate* [ERI98] and are defined using the notation represented in Figure 21.5. The diamond implies an assembly relationship. It should be noted that the connecting lines may be augmented with additional symbols (not shown) to represent cardinality. These are adapted from the entity/relationship modeling notation discussed in Chapter 12.

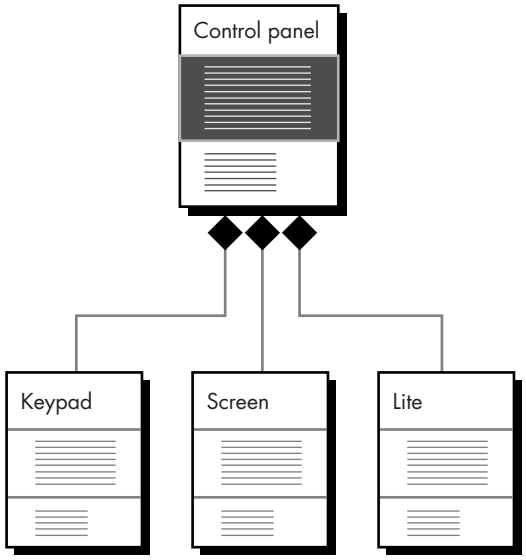


FIGURE 21.5
Class diagram for composite aggregates

Structure representations provide the analyst with a means for partitioning the CRC model and representing that partitioning graphically. The expansion of each class provides needed detail for review and for subsequent design.

21.4.4 Defining Subjects and Subsystems

A subsystem (UML package) points to a more-detailed class hierarchy.

An analysis model for a complex application may have hundreds of classes and dozens of structures. For this reason, it is necessary to define a concise representation that is a digest of the CRC and structure models just described.

When a group of all classes collaborate among themselves to accomplish a set of cohesive responsibilities, they are often referred to as *subsystems* or *packages* (in UML terminology). Subsystems or packages are abstractions that provide a reference or pointer to more detail in the analysis model. When viewed from the outside, a subsystem can be treated as a black box that contains a set of responsibilities and that has its own (outside) collaborators. A subsystem implements one or more *contracts* [WIR90] with its outside collaborators. A contract is a specific list of requests that collaborators can make of the subsystem.[8]

Subsystems can be represented with the context of CRC modeling by creating a subsystem index card. The subsystem index card indicates the name of the subsystem, the contracts that the subsystem must accommodate, and the classes or (other) subsystems that support the contract.

Packages are identical to subsystems in intent and content but are represented graphically in UML. For example, assume that the control panel for *SafeHome* is considerably more complex that the one implied by Figure 21.5, containing multiple display areas, a sophisticated key arrangement, and other features. It might be modeled as the composite aggregate structure shown in Figure 21.6. If the overall requirements model contains dozens of these structures (*SafeHome* would not), it would be difficult to absorb the entire representation at one time. By defining a package reference as shown in the figure, the entire structure can be referenced by a single icon (the file folder). Package references can be created for any structure that has multiple objects.

At the most abstract level, the OOA model would contain only package references such as those illustrated at the top of Figure 21.7. Each of the references would be expanded into a structure. Structures for the **control panel** and **sensor** objects (Figures 21.5 and 21.6) are shown in the figure; structures for **system, sensor event** and **audible alarm** would also be created.

The dashed arrows shown at the top of Figure 21.7 represent dependence relationships between the packages shown. For example, **sensor** depends on the status of the **sensor event** package. The solid arrows represent composition. In the example shown, the **system** package is composed of the **control panel, sensor,** and **audible alarm** packages.

---

8 Recall that classes interact using a client/server philosophy. In this case, the subsystem is the server and outside collaborators are clients.

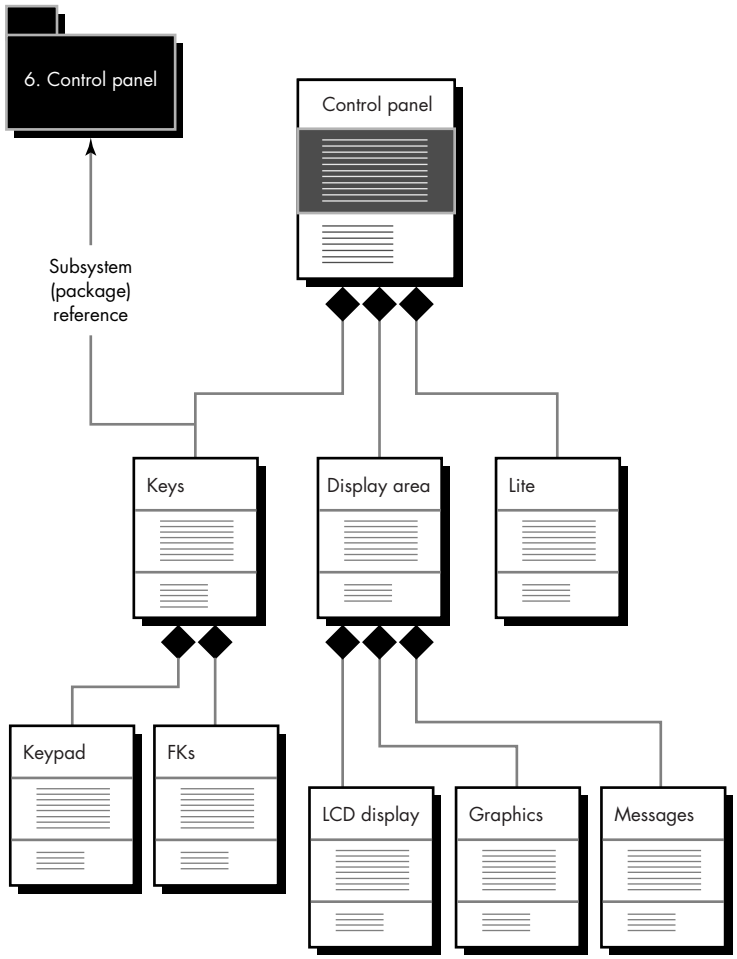


FIGURE 21.6
Package (subsystem) reference

21.5 THE OBJECT-RELATIONSHIP MODEL

The CRC modeling approach establishes the first elements of class and object relationships. The first step in establishing relationships is to understand the responsibilities for each class. The CRC model index card contains a list of responsibilities. The next step is to define those collaborator classes that help in achieving each responsibility. This establishes the "connection" between classes.

A relationship exists between any two classes that are connected.[9] Therefore, collaborators are always related in some way. The most common type of relationship is binary—a connection exists between two classes. When considered within the context of an OO system, a binary relationship has a specific direction[10] that is defined based on which class plays the role of the client and which acts as a server.

9 Other terms for relationship are *association* [RUM91] and *connection* [COA91].
10 It is important to note that this is a departure from the bidirectional nature of relationships used in data modeling (Chapter 12).

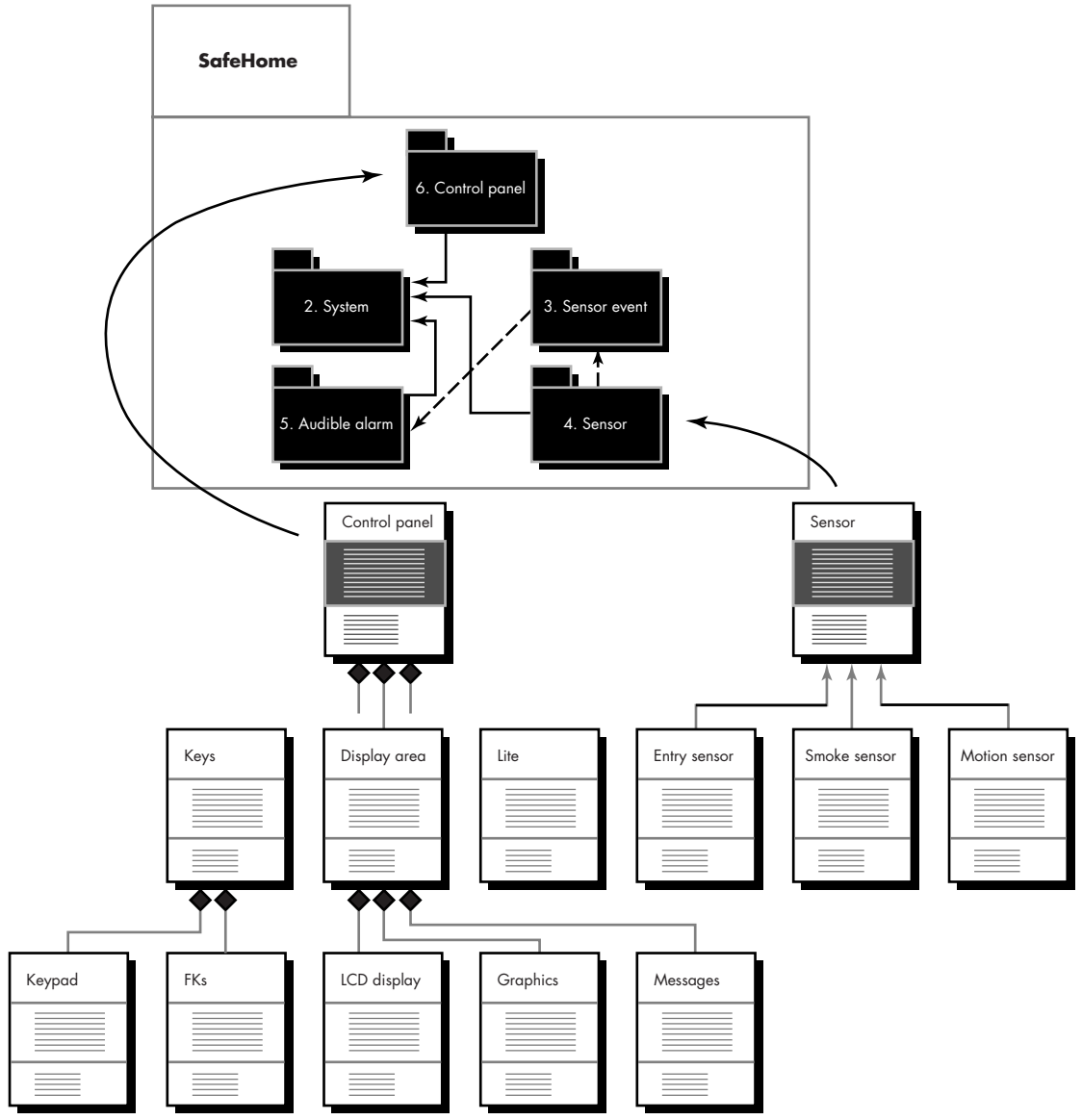


FIGURE 21.7 An analysis model with package references

Rumbaugh and his colleagues [RUM91] suggest that relationships can be derived by examining the stative verbs or verb phrases in the statement of scope or use-cases for the system. Using a grammatical parse, the analyst isolates verbs that indicate physical location or placement (next to, part of, contained in), communications (transmits to, acquires from), ownership (incorporated by, is composed of), and satisfaction of a condition (manages, coordinates, controls). These provide an indication of a relationship.

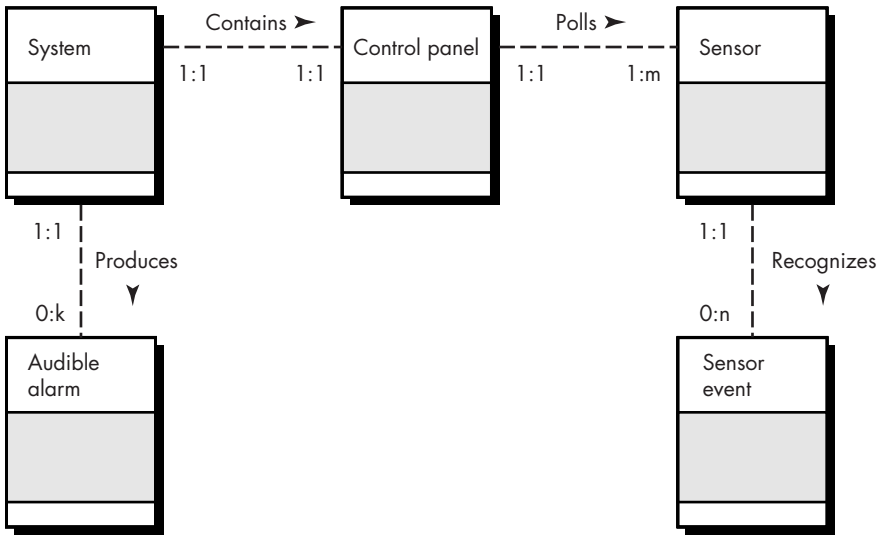


FIGURE 21.8 Relationships between objects

The Unified Modeling Language notation for the object-relationship model makes use of a symbology that has been adapted from the entity-relationship modeling techniques discussed in Chapter 12. In essence, objects are connected to other objects using named relationships. The cardinality of the connection (see Chapter 12) is specified and an overall network of relationships is established.

The object relationship model (like the entity relationship model) can be derived in three steps:

How is an object-relationship model derived?

1. **Using the CRC index cards, a network of collaborator objects can be drawn.** Figure 21.8 represents the class connections for *SafeHome* objects. First the objects are drawn, connected by unlabeled lines (not shown in the figure) that indicate some relationship exists between the connected objects.
2. **Reviewing the CRC model index card, responsibilities and collaborators are evaluated and each unlabeled connected line is named.** To avoid ambiguity, an arrow head indicates the "direction" of the relationship (Figure 21.8).
3. **Once the named relationships have been established, each end is evaluated to determine cardinality** (Figure 21.8). Four options exist: 0 to 1, 1 to 1, 0 to many, or 1 to many. For example, the *SafeHome* system contains a single control panel (the 1:1 cardinality notation indicates this). At least one sensor must be present for polling by the control panel. However, there may be many sensors present (the 1:m notation indicates this). One sensor can recognize from 0 to many sensor events (e.g., smoke is detected or a break-in has occurred).

The steps just noted continue until a complete object-relationship model has been produced.

By developing an object-relationship model, the analyst adds still another dimension to the overall analysis model. Not only are the relationships between objects identified, but all important message paths are defined (Chapter 20). In our discussion of Figure 21.7, we made reference to the arrows that connected package symbols. These are also message paths. Each arrow implies the interchange of messages among subsystems in the model.

21.6 THE OBJECT-BEHAVIOR MODEL

The CRC model and the object-relationship model represent static elements of the OO analysis model. It is now time to make a transition to the dynamic behavior of the OO system or product. To accomplish this, we must represent the behavior of the system as a function of specific events and time.

The object-behavior model indicates how an OO system will respond to external events or stimuli. To create the model, the analyst must perform the following steps:

What are the steps required to build an object-behavior model?

1. Evaluate all use-cases (Section 21.4.1) to fully understand the sequence of interaction within the system.
2. Identify events that drive the interaction sequence and understand how these events relate to specific objects.
3. Create an event trace [RUM91] for each use-case.
4. Build a state transition diagram for the system.
5. Review the object-behavior model to verify accuracy and consistency.

Each of these steps is discussed in the sections that follow.

21.6.1 Event Identification with Use-Cases

As we noted in Section 21.4.1, the use-case represents a sequence of activities that involves actors and the system. In general, an event occurs whenever an OO system and an actor (recall that an actor can be a person, a device, or even an external system) exchange information. Recalling the discussion presented in Chapter 12, it is important to note that an event is Boolean. That is, an event is *not* the information that has been exchanged but rather the fact that information has been exchanged.

A use-case is examined for points of information exchange. To illustrate, reconsider the use-case for *SafeHome* described in Section 11.2.4:

1. The homeowner observes the *SafeHome* control panel (Figure 11.2) to determine if the system is ready for input. If the system is *not ready*, the homeowner must physically close windows/doors so that the *ready* indicator is present. [A *not-ready* indicator implies that a sensor is open, i.e., that a door or window is open.]
2. The homeowner uses the keypad to key in a four-digit password. The password is compared with the valid password stored in the system. If the password is incorrect, the con-

trol panel will beep once and reset itself for additional input. If the password is correct, the control panel awaits further action.

3. The homeowner selects and keys in *stay* or *away* to activate the system. *Stay* activates only perimeter sensors (inside motion detecting sensors are deactivated). *Away* activates all sensors.

4. When activation occurs, a red alarm light can be observed by the homeowner.

The underlined portions of the use-case scenario indicate events. An actor should be identified for each event; the information that is exchanged should be noted; and any conditions or constraints should be listed.

As an example of a typical event, consider the underlined use-case phrase "homeowner uses the keypad to key in a four-digit password." In the context of the OO analysis model, the object, **homeowner,** transmits an event to the object **control panel.** The event might be called *password entered.* The information transferred is the four digits that constitute the password, but this is not an essential part of the behavioral model. It is important to note that some events have an explicit impact on the flow of control of the use-case, while others have no direct impact on the flow of control. For example, the event *password entered* does not explicitly change the flow of control of the use-case, but the results of the event *compare password* (derived from the interaction "password is compared with the valid password stored in the system") will have an explicit impact on the information and control flow of the *SafeHome* software.

Once all events have been identified, they are allocated to the objects involved. Objects can be responsible for generating events (e.g., **homeowner** generates the *password entered* event) or recognizing events that have occurred elsewhere (e.g., **control panel** recognizes the binary result of the *compare password* event).

21.6.2 State Representations

In the context of OO systems, two different characterizations of states must be considered: (1) the state of each object as the system performs its function and (2) the state of the system as observed from the outside as the system performs its function.

As you begin to identify states, focus on externally observable modes of behavior. Later, you may refine these states into behaviors that are not evident from outside the system.

The state of an object takes on both passive and active characteristics [CHA93]. A *passive state* is simply the current status of all of an object's attributes. For example, the passive state of the aggregate object **player** (in the video game application discussed earlier) would include the current **position** and **orientation** attributes of **player** as well as other features of player that are relevant to the game (e.g., an attribute that indicates **magic wishes remaining**). The *active state* of an object indicates the current status of the object as it undergoes a continuing transformation or processing. The object **player** might have the following active states: *moving, at rest, injured, being cured; trapped, lost,* and so forth. An event (sometimes called a *trigger*) must occur to force an object to make a transition from one active state to another. One component of

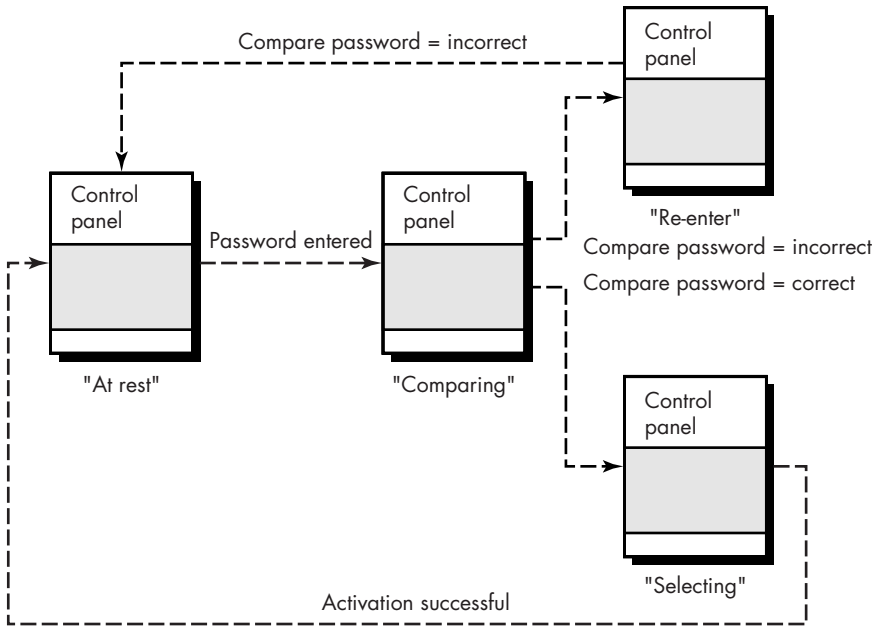


FIGURE 21.9
A representation of active state transitions

an object-behavior model is a simple representation of the active states for each object and the events (triggers) that cause changes between these active states. Figure 21.9 illustrates a simple representation of active states for the **control panel** object in the *SafeHome* system.

Each arrow shown in Figure 21.9 represents a transition from one active state of an object to another. The labels shown for each arrow represent the event that triggers the transition. Although the active state model provides useful insight into the "life history" of an object, it is possible to specify additional information to provide more depth in understanding the behavior of an object. In addition to specifying the event that causes the transition to occur, the analyst can specify a guard and an action [CHA93]. A *guard* is a Boolean condition that must be satisfied in order for the transition to occur. For example, the guard for the transition from the "at rest" state to the "comparing state" in Figure 21.9 can be determined by examining the use-case:

if (password input = 4 digits) then make transition to comparing state;

In general, the guard for a transition usually depends upon the value of one or more attributes of an object. In other words, the guard depends on the passive state of the object.

An *action* occurs concurrently with the state transition or as a consequence of it and generally involves one or more operations (responsibilities) of the object. For example, the action connected to the *password entered* event (Figure 21.9) is an operation that accesses a password object and performs a digit-by-digit comparison to validate the entered password.

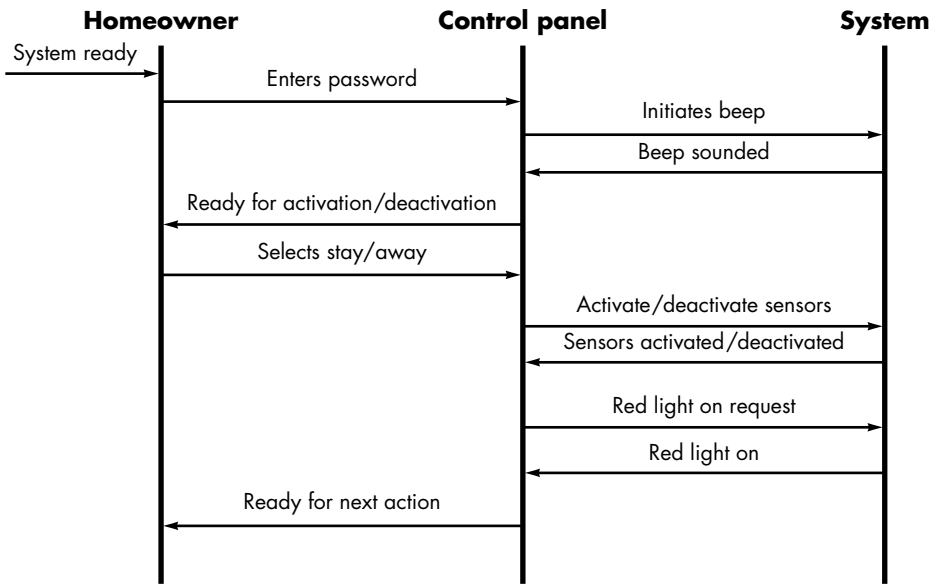


FIGURE 21.10
A partial event trace for SafeHome

The second type of behavioral representation for OOA considers a state representation for the overall product or system. This representation encompasses a simple *event trace model* [RUM91] that indicates how events cause transitions from object to object and a state transition diagram that depicts the processing behavior of each object.

KEY POINT
A transition from one state to another requires that an event occur. Events are Boolean in nature and often occur when objects communicate with one another.

Once events have been identified for a use-case, the analyst creates a representation of how events cause flow from one object to another. Called an *event trace,* this representation is a shorthand version of the use-case. It represents key objects and the events that cause behavior to flow from object to object.

Figure 21.10 illustrates a partial event trace for the *SafeHome* system. Each of the arrows represents an event (derived from a use-case) and indicates how the event channels behavior between *SafeHome* objects. The first event, *system ready,* is derived from the external environment and channels behavior to the **homeowner** object. The homeowner enters a password. The event initiates *beep* and "beep sounded" and indicates how behavior is channeled if the password is invalid. A valid password results in flow back to **homeowner.** The remaining events and traces follow the behavior as the system is activated or deactivated.

Once a complete event trace has been developed, all of the events that cause transitions between system objects can be collated into a set of input events and output events (from an object). This can be represented using an event flow diagram [RUM91]. All events that flow into and out of an object are noted as shown in Figure 21.11. A state transition diagram (Chapter 12) can then be developed to represent the behavior associated with responsibilities for each class.

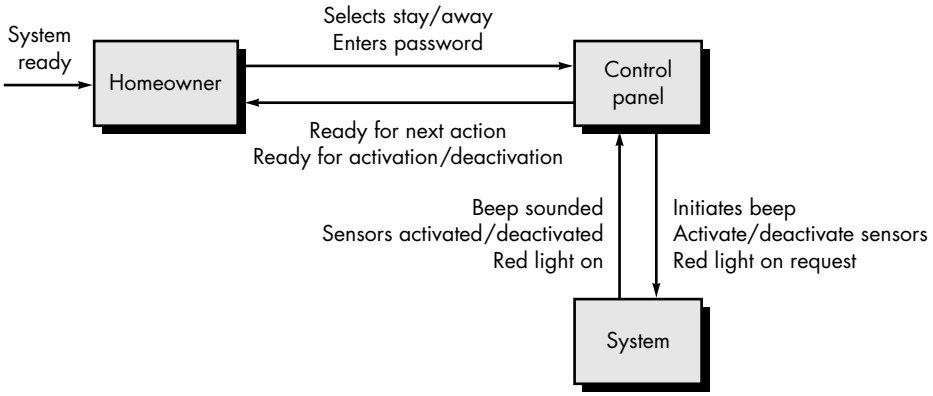


FIGURE 21.11 A partial event flow diagram for SafeHome

UML uses a combination of state diagrams, sequence diagrams, collaboration diagrams, and activity diagrams to represent the dynamic behavior of the objects and classes that have been identified as part of the analysis model. A complete discussion of these graphical representations and the language descriptions that underlie them is beyond the scope of this book. The interested reader should see [BOO99], [BEN99], [ALH98], and [ERI98] for additional detail.

21.7 SUMMARY

Object-oriented analysis methods enable a software engineer to model a problem by representing both static and dynamic characteristics of classes and their relationships as the primary modeling components. Like earlier OO analysis methods, the Unified Modeling Language builds an analysis model that has the following characteristics: (1) representation of classes and class hierarchies, (2) creation of object-relationship models, and (3) derivation of object-behavior models.

Analysis for object-oriented systems occurs at many different levels of abstraction. At the business or enterprise level, the techniques associated with OOA can be coupled with a business process engineering approach. This technique is often called domain analysis. At an application level, the object model focuses on specific customer requirements as those requirements affect the application to be built.

The OOA process begins with the definition of use-cases—scenarios that describe how the OO system is to be used. The class-responsibility-collaborator modeling technique is then applied to document classes and their attributes and operations. It also provides an initial view of the collaborations that occur among objects. The next step in the OOA process is classification of objects and the creation of a class hierarchy. Subsystems (packages) can be used to encapsulate related objects. The object-relationship model provides an indication of how classes are connected to one another, and the object-behavior model indicates the behavior of individual objects and the overall behavior of the OO system.

REFERENCES

[ALH98] Alhir, S.S., *UML in a Nutshell,* O'Reilly & Associates, 1998.

[AMB95] Ambler, S., "Using Use-Cases," *Software Development,* July 1995, pp. 53–61.

[ARA89] Arango, G. and R. Prieto-Diaz, "Domain Analysis: Concepts and Research Directions," *Domain Analysis: Acquisition of Reusable Information for Software Construction,* (Arango, G. and R. Prieto-Diaz, eds.), IEEE Computer Society Press, 1989.

[BEN99] Bennett, S., S. McRobb, and R. Farmer, *Object Oriented System Analysis and Design Using UML,* McGraw-Hill, 1999.

[BER93] Berard, E.V., *Essays on Object-Oriented Software Engineering,* Addison-Wesley, 1993.

[BOO94] Booch, G., *Object-Oriented Analysis and Design,* 2nd ed., Benjamin Cummings, 1994.

[BOO99] Booch, G., I. Jacobson, J. Rumbaugh, *The Unified Modeling Language User Guide,* Addison-Wesley, 1999.

[CAR98] Carmichael, A., *Developing Business Objects,* SIGS Books, 1998).

[CHA93] De Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development,* Addison-Wesley, 1993.

[COA91] Coad, P. and E. Yourdon, *Object-Oriented Analysis,* 2nd ed., Prentice-Hall, 1991.

[EEL98] Eeles, P. and O. Sims, *Building Business Objects,* Wiley, 1998.

[ERI98] Eriksson, H.E. and M. Penker, *UML Toolkit,* Wiley, 1998.

[FIC92] Fichman, R.G. and C.F. Kemerer, "Object-Oriented and Conventional Analysis and Design Methodologies," *Computer,* vol. 25, no. 10, October 1992, pp. 22–39.

[FIN96] Fingar, P., *The Blueprint for Business Objects,* Cambridge University Press, 1996.

[FIR93] Firesmith, D.G., *Object-Oriented Requirements Analysis and Logical Design,* Wiley, 1993.

[GRA94] Graham, I., *Object-Oriented Methods,* Addison-Wesley, 1994.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[JAC99] Jacobson, I., G. Booch, J. Rumbaugh, *Unified Software Development Process,* Addison-Wesley, 1999.

[MAT94] Mattison, R., *The Object-Oriented Enterprise,* McGraw-Hill, 1994.

[MON92] Monarchi, D.E. and G.I. Puhr, "A Research Typology for Object-Oriented Analysis and Design," *CACM,* vol. 35, no. 9, September 1992, pp. 35–47.

[RUM91] Rumbaugh, J., et al., *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

[RUM99] Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.

[SUL94] Sullo, G.C., *Object Engineering,* Wiley, 1994.

[TAY95] Taylor, D.A., *Business Engineering with Object Technology,* Wiley, 1995.

[WIR90] Wirfs-Brock, R., B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software,* Prentice-Hall, 1990.

PROBLEMS AND POINTS TO PONDER

21.1. Obtain one or more books dedicated to the Unified Modeling Language and compare it to structured analysis (Chapter 12) using the modeling dimensions proposed by Fichman and Kemerer [FIC92] in Section 21.1.1.

21.2. Develop a classroom presentation on one static or dynamic modeling diagram used in UML. Present the diagram in the context of a simple example, but provide enough detail to demonstrate most important aspects of the diagrammatic form.

21.3. Conduct an abbreviated domain analysis for one of the following areas:

a. A university student record-keeping system.

b. An e-commerce application (e.g., clothes, books, electronic gear).

c. Customer service for a bank.

d. A video game developer.

e. An application area suggested by your instructor.

Be sure to isolate classes that can be used for a number of applications in the domain.

21.4. In your own words describe the difference between static and dynamic views of an OO system.

21.5. Write a use-case for the *SafeHome* system discussed in this book. The use-case should address the scenario required to define a security zone. A security zone encompasses a set of sensors can be addressed, activated, and deactivated as a set rather than individually. As many as ten security zones can be defined. Be creative here but stay within the bounds of the *SafeHome* control panel as it is defined earlier in the book.

21.6. Develop a set of use-cases for the PHTRS system introduced in Problem 12.13. You'll have to make a number of assumptions about the manner in which a user interacts with this system.

21.7. Develop a set of use-cases for any one of the following applications:

a. Software for a general-purpose personal digital assistant.

b. Software for a video game of your choosing.

c. Software that sits inside a climate control system for a car.

d. Software for a navigation system for a car.

e. A system (product) suggested by your instructor.

Do a few hours of research on the application area and conduct a FAST meeting (Chapter 11) with your fellow students to develop basic requirements (your instructor will help you coordinate this).

21.8. Develop a complete set of CRC model index cards on the product or system you chose as part of Problem 21.7.

21.9. Conduct a review of the CRC index cards with your colleagues. How many additional classes, responsibilities, and collaborators were added as a consequence of the review?

21.10. Develop a class hierarchy for the product or system you chose as part of Problem 21.7.

21.11. Develop a set of subsystems (packages) for the product or system you chose as part of Problem 21.7.

21.12. Develop an object-relationship model for the product or system you chose as part of Problem 21.7.

21.13. Develop an object-behavior model for the product or system you chose as part of Problem 21.7. Be sure to list all events, provide an event trace, develop an event flow diagram, and define state diagram for each class.

21.14. In your own words, describe how collaborators for a class are determined.

21.15. What strategy would you propose for defining subsystems for a collection of classes?

21.16 What role does cardinality play in the development of an object-relationship model?

21.17. What is the difference between an active and a passive state for an object?

FURTHER READINGS AND INFORMATION SOURCES

Use-cases form the foundation of object-oriented analysis, regardless of the OOA method that is chosen. Books by Rosenberg and Scott (*Use Case Driven Object Modeling with UML: A Practical Approach,* Addison-Wesley, 1999); Schneider, Winters, and Jacobson (*Applying Use Cases: A Practical Guide,* Addison-Wesley, 1998); and Texel and Williams (*Use Cases Combined With Booch/OMT/UML: Process and Products,* Prentice-Hall, 1997) provide worthwhile guidance in the creation and use of this important requirements elicitation and representation mechanism.

Virtually every recent book published on object-oriented analysis and design emphasizes UML. Those serious about applying UML in their work should acquire [BOO99], [RUM99], and [JAC99]. In addition, the following books are representative of dozens written on UML technology:

Douglass, B., *Real-Time UML: Developing Efficient Objects for Embedded Systems,* Addison-Wesley, 1999.

Fowler, M. and K. Scott, *UML Distilled,* 2nd ed., Addison-Wesley, 2000.

Odell, J.J. and M. Fowler, *Advanced Object-Oriented Analysis and Design Using UML,* SIGS Books, 1998.

Oestereich, B., *Developing Software with UML: Object-Oriented Analysis and Design in Practice,* Addison-Wesley, 1999.

A wide variety of information sources on object-oriented analysis and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to OOA can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/OOA.mhtml

CHAPTER

22 OBJECT-ORIENTED DESIGN

KEY CONCEPTS

Object-oriented design transforms the analysis model created using object-oriented analysis (Chapter 21) into a design model that serves as a blueprint for software construction. Yet, the job of the software designer can be daunting. Gamma and his colleagues [GAM95] provide a reasonably accurate picture of OOD when they state:

> Designing object-oriented software is hard, and designing reusable object-oriented software is even harder. You must find pertinent objects, factor them into classes at the right granularity, define class interfaces and inheritance hierarchies, and establish key relationships among them. Your design should be specific to the problem at hand but also general enough to address future problems and requirements. You also want to avoid redesign, or at least minimize it. Experienced object-oriented designers will tell you that a reusable and flexible design is difficult if not impossible to get "right" the first time. Before a design is finished, they usually try to reuse it several times, modifying it each time.

Unlike conventional software design methods, OOD results in a design that achieves a number of different levels of modularity. Major system components are organized into subsystems, a system-level "module." Data and the operations that manipulate the data are encapsulated into objects—a modular form

QUICK LOOK

What is it? The design of object-oriented software requires the definition of a multilayered software architecture, the specification of subsystems that perform required functions and provide infrastructure support, a description of objects (classes) that form the building blocks of the system, and a description of the communication mechanisms that allow data to flow between layers, subsystems, and objects. Object-oriented design accomplishes all of these things.

Who does it? OOD is performed by a software engineer.

Why is it important? An object-oriented system draws upon class definitions that are derived from the analysis model. Some of these definitions will have to be built from scratch, but many others may be reused if appropriate design patterns are recognized. OOD establishes a design blueprint that enables a software engineer to define the OO architecture in a manner that maximizes reuse, thereby improving development speed and end-product quality.

What are the steps? OOD is divided into two major activities: system design and object design. System design creates the product architecture, defining a series of "layers" that accomplish specific system functions and identifying the classes that are encapsulated by subsystems that reside at each layer. In addition, system design considers the specification of three components: the user interface, data management functions, and task

QUICK LOOK

management facilities. Object design focuses on the internal detail of individual classes, defining attributes, operations, and message detail.

What is the work product? An OO design model encompasses software architecture, user interface description, data management components, task management facilities, and detailed descriptions of each class to be used in the system.

How do I ensure that I've done it right? At each stage, the elements of the object-oriented design model are reviewed for clarity, correctness, completeness, and consistency with customer requirements and with one another.

that is the building block of an OO system. In addition, OOD must describe the specific data organization of attributes and the procedural detail of each individual operation. These represent data and algorithmic pieces of an OO system and are contributors to overall modularity.

The unique nature of object-oriented design lies in its ability to build upon four important software design concepts: abstraction, information hiding, functional independence, and modularity (Chapter 13). All design methods strive for software that exhibits these fundamental characteristics, but only OOD provides a mechanism that enables the designer to achieve all four without complexity or compromise.

Object-oriented design, object-oriented programming, and object-oriented testing are construction activities for OO systems. In this chapter, we consider the first step in construction.

22.1 DESIGN FOR OBJECT-ORIENTED SYSTEMS

In Chapter 13, we introduced the concept of a design pyramid for conventional software. Four design layers—data, architectural, interface, and component level—were defined and discussed. For object-oriented systems, we can also define a design pyramid, but the layers are a bit different. Referring to Figure 22.1, the four layers of the OO design pyramid are

Quote:

"In design, we shape the system and find its form . . ."

Ivar Jacobson, Grady Booch, and James Rumbaugh

The subsystem layer contains a representation of each of the subsystems that enable the software to achieve its customer-defined requirements and to implement the technical infrastructure that supports customer requirements.

The class and object layer contains the class hierarchies that enable the system to be created using generalizations and increasingly more targeted specializations. This layer also contains representations of each object.

The message layer contains the design details that enable each object to communicate with its collaborators. This layer establishes the external and internal interfaces for the system.

The responsibilities layer contains the data structure and algorithmic design for all attributes and operations for each object.

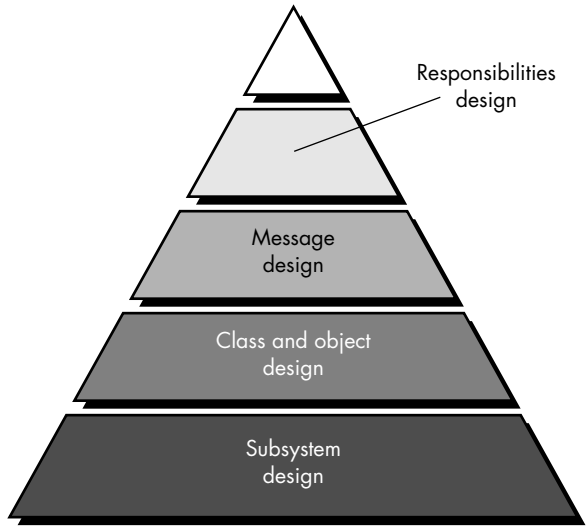


FIGURE 22.1
The OO design pyramid

The design pyramid focuses exclusively on the design of a specific product or system. It should be noted, however, that another "layer" of design exists, and this layer forms the foundation on which the pyramid rests. The foundation layer focuses on the design of *domain objects* (called *design patterns* later in this chapter). Domain objects play a key role in building the infrastructure for the OO system by providing support for human/computer interface activities, task management, and data management. Domain objects can also be used to flesh out the design of the application itself.

22.1.1 Conventional vs. OO Approaches

Conventional approaches to software design apply a distinct notation and set of heuristics to map the analysis model into a design model. Recalling Figure 13.1, each element of the conventional analysis model maps into one or more layers of the design model. Like conventional software design, OOD applies data design when attributes are represented, interface design when a messaging model is developed, and component-level (procedural) design for the design of operations. It is important to note that the architecture of an OO design has more to do with the collaborations among objects than with the flow of control between components of the system.

Although similarity between the conventional and OO design models does exist, we have chosen to rename the layers of the design pyramid to reflect more accurately the nature of an OO design. Figure 22.2 illustrates the relationship between the OO analysis model (Chapter 21) and design model that will be derived from it.[1]

1 It is important to note that the derivation is not always straightforward. For further discussion, see [DAV95].

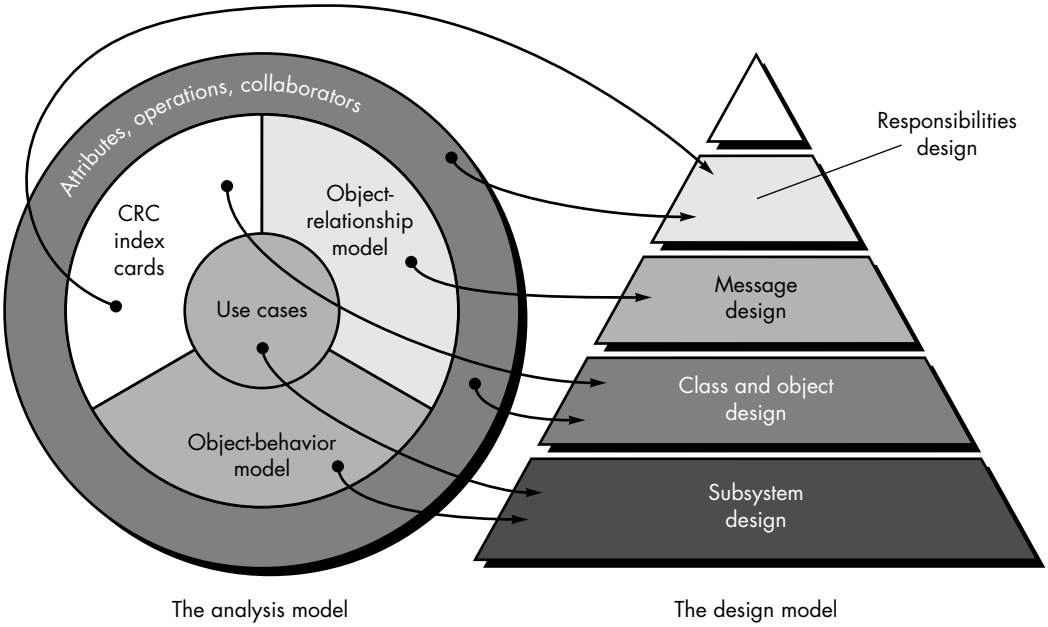


FIGURE 22.2 Translating an OOA model into an OOD model

The subsystem design is derived by considering overall customer requirements (represented with use-cases) and the events and states that are externally observable (the object-behavior model). Class and object design is mapped from the description of attributes, operations, and collaborations contained in the CRC model. Message design is driven by the object-relationship model, and responsibilities design is derived using the attributes, operations, and collaborations described in the CRC model.

Fichman and Kemerer [FIC92] suggest ten design modeling components that may be used to compare various conventional and object-oriented design methods:

? What criteria can be used to compare conventional and OOD methods?

1. Representation of hierarchy of modules.
2. Specification of data definitions.
3. Specification of procedural logic.
4. Indication of end-to-end processing sequences.
5. Representation of object states and transitions.
6. Definition of classes and hierarchies.
7. Assignment of operations to classes.
8. Detailed definition of operations.
9. Specification of message connections.
10. Identification of exclusive services.

Because many conventional and object-oriented design approaches are available, it is difficult to develop a generalized comparison between the two methods. It can be stated, however, that modeling dimensions 5 through 10 are not supported using structured design (Chapter 14) or its derivatives.

22.1.2 Design Issues

Bertrand Meyer [MEY90] suggests five criteria for judging a design method's ability to achieve modularity and relates these to object-oriented design:

A discussion that addresses the question "What makes a good object-oriented design?" can be found at **www.kinetica.com/ootips/ood-principles.html**

- *Decomposability*—the facility with which a design method helps the designer to decompose a large problem into subproblems that are easier to solve.
- *Composability*—the degree to which a design method ensures that program components (modules), once designed and built, can be reused to create other systems.
- *Understandability*—the ease with which a program component can be understood without reference to other information or other modules.
- *Continuity*—the ability to make small changes in a program and have these changes manifest themselves with corresponding changes in just one or a very few modules.
- *Protection*—an architectural characteristic that will reduce the propagation of side effects if an error does occur in a given module.

? What basic principles guide us in the design of modular architectures?

From these criteria, Meyer [MEY90] suggests five basic design principles that can be derived for modular architectures: (1) linguistic modular units, (2) few interfaces, (3) small interfaces (weak coupling), (4) explicit interfaces, and (5) information hiding.

Modules are defined as *linguistic modular units* when they "correspond to syntactic units in the language used" [MEY90]. That is, the programming language to be used should be capable of supporting the modularity defined directly. For example, if the designer creates a subroutine, any of the older programming languages (e.g., FORTRAN, C, Pascal) could implement it as a syntactic unit. But if a package that contains data structures and procedures and identifies them as a single unit were defined, a language such as Ada (or another object-oriented language) would be necessary to directly represent this type of component in the language syntax.

To achieve low coupling (a design concept introduced in Chapter 13), the number of interfaces between modules should be minimized ("few interfaces") and the amount of information that moves across an interface should be minimized ("small interfaces"). Whenever components do communicate, they should do so in an obvious and direct way ("explicit interfaces"). For example, if component X and component Y communicate through a global data area (what we called *common coupling* in Chapter 13), they violate the principle of explicit interfaces because the communication between the components is not obvious to an outside observer. Finally, we

achieve the principle of information hiding when all information about a component is hidden from outside access, unless that information is specifically defined as *public information.*

The design criteria and principles presented in this section can be applied to any design method (e.g., we can apply them to structured design). As we will see, however, the object-oriented design method achieves each of the criteria more efficiently than other approaches and results in modular architectures that allow us to meet each of the modularity criteria most effectively.

22.1.3 The OOD Landscape

As we noted in Chapter 21, a wide variety of object-oriented analysis and design methods were proposed and used during the 1980s and 1990s. These methods established the foundation for modern OOD notation, design heuristics, and models. A brief overview of the most important early OOD methods follows:

Quote:

"There is no reason why the transition from requirements to design should be any easier in software engineering than it is in any other engineering discipline. Design is hard."

Alan Davis

The Booch method. As we noted in Chapter 21, the Booch method [BOO94] encompasses both a "micro development process" and a "macro development process." In the design context, macro development encompasses an architectural planning activity that clusters similar objects in separate architectural partitions, layers objects by level of abstraction, identifies relevant scenarios, creates a design prototype, and validates the design prototype by applying it to usage scenarios. Micro development defines a set of "rules" that govern the use of operations and attributes and the domain-specific policies for memory management, error handling, and other infrastructure functions; develops scenarios that describe the semantics of the rules and policies; creates a prototype for each policy; instruments and refines the prototype; and reviews each policy so that it "broadcasts its architectural vision" [BOO94].

The Rumbaugh method. The *object modeling technique* [RUM91] encompasses a design activity that encourages design to be conducted at two different levels of abstraction. *System design* focuses on the layout for the components that are needed to construct a complete product or system. The analysis model is partitioned into subsystems, which are then allocated to processors and tasks. A strategy for implementing data management is defined and global resources and the control mechanisms required to access them are identified.

Object design emphasizes the detailed layout of an individual object. Operations are selected from the analysis model and algorithms are defined for each operation. Data structures that are appropriate for attributes and algorithms are represented. Classes and class attributes are designed in a manner that optimizes access to data and improves computational efficiency. A messaging model is created to implement the object relationships (associations).

The Jacobson method. The design activity for OOSE (object-oriented software engineering) [JAC92] is a simplified version of the proprietary *objectory method,* also developed by Jacobson. The design model emphasizes traceability to the OOSE analysis model. First, the idealized analysis model is adapted to fit the real world environment. Then primary design objects, called *blocks,*[2] are created and categorized as interface blocks, entity blocks, and control blocks. Communication between blocks during execution is defined and the blocks are organized into subsystems.

The Coad and Yourdon method. The Coad and Yourdon method for OOD [COA91] was developed by studying how "effective object-oriented designers" do their design work. The design approach addresses not only the application but also the infrastructure for the application and focuses on the representation of four major system components: the problem domain component, the human interaction component, the task management component, and the data management component.

Although it is not nearly as robust as UML, the Wirfs-Brock method has a simple elegance that makes it an interesting alternative approach to OOD.

The Wirfs-Brock method. Wirfs-Brock, Wilkerson, and Weiner [WIR90] define a continuum of technical tasks in which analysis leads seamlessly into design. *Protocols*[3] for each class are constructed by refining contracts between objects. Each operation (responsibility) and protocol (interface design) is designed at a level of detail that will guide implementation. Specifications for each class (defining private responsibilities and detail for operations) and each subsystem (identifying all encapsulated classes and the interaction between subsystems) are developed.

Although the terminology and process steps for each of these OOD methods differ, the overall OOD processes are reasonably consistent. To perform object-oriented design, a software engineer should perform the following generic steps:

A set of generic steps are applied during OOD, regardless of the design method that is chosen.

1. Describe each subsystem and allocate it to processors and tasks.
2. Choose a design strategy for implementing data management, interface support, and task management.
3. Design an appropriate control mechanism for the system.
4. Perform object design by creating a procedural representation for each operation and data structures for class attributes.
5. Perform message design using collaborations between objects and object relationships.
6. Create the messaging model.
7. Review the design model and iterate as required.

---

2 A *block* is the design abstraction that allows for the representation of an aggregate object.

3 A *protocol* is a formal description of the messages to which a class will respond.

It is important to note that the design steps discussed in this section are iterative. That is, they may be executed incrementally, along with additional OOA activities, until a completed design is produced.

22.1.4 A Unified Approach to OOD

In Chapter 21, we noted that Grady Booch, James Rumbaugh, and Ivar Jacobson combined the best features of their individual object-oriented analysis and design methods into a unified method. The result, called the *Unified Modeling Language* has become widely used throughout the industry.[4]

An extensive tutorial and listing of UML resources including tools, papers, and examples can be found at **mini.net/cetus/oo_uml.html**

During analysis modeling (Chapter 21), the user model and structural model views are represented. These provide insight into the usage scenarios for the system (providing guidance for behavioral modeling) and establish a foundation for the implementation and environment model views by identifying and describing the static structural elements of the system.

UML is organized into two major design activities: system design and object design. The primary objective of UML *system design* is to represent the software architecture. Bennett, McRobb, and Farmer [BEN99] discuss this issue in the following way:

> In terms of object-oriented development, the conceptual architecture is concerned with the structure of the static class model and the connections between components of the model. The module architecture describes the way the system is divided into subsystems or modules and how they communicate by exporting and importing data. The code architecture defines how the program code is organized into files and directories and grouped into libraries. The execution architecture focuses on the dynamic aspects of the system and the communication between components as tasks and operations execute.

System design focuses on software architecture and the definition of subsystems. Object design describes objects at a level of detail that can be implemented in a programming language.

The definition of the "subsystems" noted by Bennett et al. is a primary concern during UML system design.

UML *object design* focuses on a description of objects and their interactions with one another. A detailed specification of attribute data structures and a procedural design of all operations are created during object design. The visibility[5] for all class attributes is defined and interfaces between objects are elaborated to define the details of a complete messaging model.

System and object design in UML are extended to consider the design of user interfaces, data management with the system to be built, and task management for the subsystems that have been specified. User interface design in UML draws on the same concepts and principles discussed in Chapter 15. The user model view drives the user interface design process, providing a scenario that is elaborated iteratively to become a set of interface classes.[6]

---

4 Booch, Rumbaugh, and Jacobson have written a set of three definitive books on UML. The interested reader should see [BOO99], [RUM99], and [JAC99].

5 *Visibility* indicates whether an attribute is public (available across all instantiations of the class), private (available only for the class that specifies it), or protected (an attribute that may be used by the class that specifies it and its subclasses).

6 Today, most interface classes are part of a library of reusable software components. This expedites the design and implementation of GUIs.

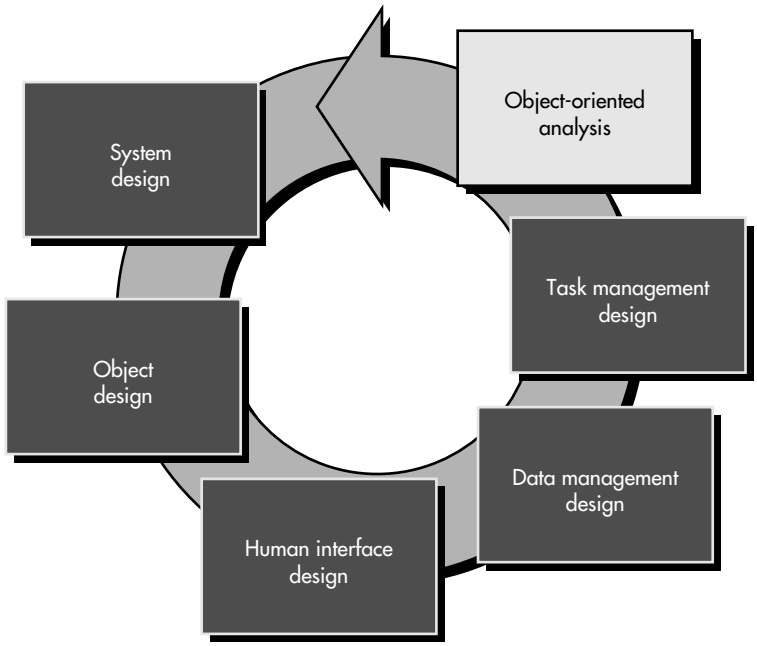


FIGURE 22.3
Process flow for OOD

Data management design establishes a set of classes and collaborations that allow the system (product) to manage persistent data (e.g., files and databases). Task management design establishes the infrastructure that organizes subsystems into tasks and then manages task concurrency. The process flow for design is illustrated in Figure 22.3.[7]

Throughout the UML design process, the user model view and structure model view are elaborated into the design representation outlined above. This elaboration activity is discussed in the sections that follow.

22.2 THE SYSTEM DESIGN PROCESS

System design develops the architectural detail required to build a system or product. The system design process encompasses the following activities:

What are the steps of the system design process?

- Partition the analysis model into subsystems.
- Identify concurrency that is dictated by the problem.
- Allocate subsystems to processors and tasks.
- Develop a design for the user interface.
- Choose a basic strategy for implementing data management.
- Identify global resources and the control mechanisms required to access them.

7 Recall that OOA is an iterative activity. It is entirely possible that the analysis model will be revised as a consequence of design work.

- Design an appropriate control mechanism for the system, including task management.
- Consider how boundary conditions should be handled.
- Review and consider trade-offs.

In the sections that follow, design activities related to each of these steps are considered in more detail.

22.2.1 Partitioning the Analysis Model

The concepts of coupling and cohesion (Chapter 13) can be applied at the subsystem level. Strive to achieve good functional independence as you design subsystems.

One of the fundamental analysis principles (Chapter 11) is partitioning. In OO system design, we partition the analysis model to define cohesive collections of classes, relationships, and behavior. These design elements are packaged as a subsystem.

In general, all of the elements of a subsystem share some property in common. They all may be involved in accomplishing the same function; they may reside within the same product hardware, or they may manage the same class of resources. Subsystems are characterized by their responsibilities; that is, a subsystem can be identified by the services that it provides [RUM91]. When used in the OO system design context, a service is a collection of operations that perform a specific function (e.g., managing word-processor files, producing a three-dimensional rendering, translating an analog video signal into a compressed digital image).

As subsystems are defined (and designed), they should conform to the following design criteria:

? What criteria guide us in the design of subsystems?

- The subsystem should have a well-defined interface through which all communication with the rest of the system occurs.
- With the exception of a small number of "communication classes," the classes within a subsystem should collaborate only with other classes within the subsystem.
- The number of subsystems should be kept low.
- A subsystem can be partitioned internally to help reduce complexity.

When two subsystems communicate with one another, they can establish a client/server link or a peer-to-peer link [RUM91]. In a client/server link, each of the subsystems takes on one of the roles implied by client and server. Service flows from server to client in only one direction. In a peer-to-peer link, services may flow in either direction.

When a system is partitioned into subsystems, another design activity, called *layering,* also occurs. Each layer [BUS96] of an OO system contains one or more subsystems and represents a different level of abstraction of the functionality required to accomplish system functions. In most cases, the levels of abstraction are determined by the degree to which the processing associated with a subsystem is visible to an end-user.

For example, a four-layer architecture might might include (1) a presentation layer (the subsystems associated with the user interface), (2) an application layer (the subsystems that perform the processing associated with the application), (3) a data formatting layer (the subsystems that prepare the data for processing), and (4) a database layer (the subsystems associated with data management). Each layer moves deeper into the system, representing increasingly more environment-specific processing.

Buschmann and his colleagues [BUS96] suggest the following design approach for layering:

How do I create a layered design?

1. Establish layering criteria. That is, decide how subsystems will be grouped in a layered architecture.
2. Determine the number of layers. Too many introduce unnecessary complexity; too few may harm functional independence.
3. Name the layers and allocate subsystems (with their encapsulated classes) to a layer. Be certain that communication between subsystems (classes) on one layer and other subsystems (classes) at another layer follow the design philosophy for the architecture.[8]
4. Design interfaces for each layer.
5. Refine the subsystems to establish the class structure for each layer.
6. Define the messaging model for communication between layers.
7. Review the layer design to ensure that coupling between layers is minimized (a client/server protocol can help accomplish this).
8. Iterate to refine the layered design.

22.2.2 Concurrency and Subsystem Allocation

ADVICE

In most cases, a multiprocessor implementation increases complexity and technical risk. Whenever possible, choose the simplest processor architecture that will get the job done.

The dynamic aspect of the object-behavior model provides an indication of concurrency among classes (or subsystems). If classes (or subsystems) are not active at the same time, there is no need for concurrent processing. This means that the classes (or subsystems) can be implemented on the same processor hardware. On the other hand, if classes (or subsystems) must act on events asynchronously and at the same time, they are viewed as concurrent. When subsystems are concurrent, two allocation options exist: (1) Allocate each subsystem to an independent processor or (2) allocate the subsystems to the same processor and provide concurrency support through operating system features.

Concurrent tasks are defined [RUM91] by examining the state diagram for each object. If the flow of events and transitions indicates that only a single object is active at any one time, a thread of control has been established. The thread of control

8 In a *closed* architecture, messages from one layer may be sent only to the adjacent lower layer. In an *open* architecture, messages may be sent to any lower layer.

continues even when one object sends a message to another object, as long as the first object waits for a response. If, however, the first object continues processing after sending a message, the thread of control splits.

Tasks in an OO system are designed by isolating threads of control. For example, while the *SafeHome* security system is monitoring its sensors, it can also be dialing the central monitoring station for verification of connection. Since the objects involved in both of these behaviors are active at the same time, each represents a separate thread of control and each can be defined as a separate task. If the monitoring and dialing activities occur sequentially, a single task could be implemented.

To determine which of the processor allocation options is appropriate, the designer must consider performance requirements, costs, and the overhead imposed by interprocessor communication.

22.2.3 The Task Management Component

Coad and Yourdon [COA91] suggest the following strategy for the design of the objects that manage concurrent tasks:

- The characteristics of the task are determined.
- A coordinator task and associated objects are defined.
- The coordinator and other tasks are integrated.

Quote:

"Discipline and focused awareness . . . contribute to the act of creation."

John Poppy

The characteristics of a task are determined by understanding how the task is initiated. Event-driven and clock-driven tasks are the most commonly encountered. Both are activated by an interrupt, but the former receives an interrupt from some outside source (e.g., another processor, a sensor) while that latter is governed by a system clock.

In addition to the manner in which a task is initiated, the priority and criticality of the task must also be determined. High-priority tasks must have immediate access to system resources. High-criticality tasks must continue to operate even if resource availability is reduced or the system is operating in a degraded state.

Once the characteristics of the task have been determined, object attributes and operations required to achieve coordination and communication with other tasks are defined. The basic task template (for a task object) takes the form [COA91]

Task name—the name of the object
Description—a narrative describing the purpose of the object
Priority—task priority (e.g., low, medium, high)
Services—a list of operations that are responsibilities of the object
Coordinates by—the manner in which object behavior is invoked
Communicates via—input and output data values relevant to the task

This template description can then be translated into the standard design model (incorporating representation of attributes and operations) for the task object(s).

22.2.4 The User Interface Component

Although the user interface component is implemented within the context of the problem domain, the interface itself represents a critically important subsystem for most modern applications. The OO analysis model (Chapter 21) contains usage scenarios (called *use-cases*) and a description of the roles that users play (called *actors*) as they interact with the system. These serve as input to the user interface design process.

XRef

Most of the classes necessary to build a modern interface already exist and are available to the designer. The design of the interface follows the approach defined in Chapter 15.

Once the actor and its usage scenario are defined, a command hierarchy is identified. The command hierarchy defines major system menu categories (the menu bar or tool palette) and all subfunctions that are available within the context of a major system menu category (the menu windows). The command hierarchy is refined iteratively until every use-case can be implemented by navigating the hierarchy of functions.

Because a wide variety of user interface development environments already exist, the design of GUI elements is not necessary. Reusable classes (with appropriate attributes and operations) already exist for windows, icons, mouse operations, and a wide variety of other interaction functions. The implementer need only instantiate objects that have appropriate characteristics for the problem domain.

22.2.5 The Data Management Component

Data management encompasses two distinct areas of concern: (1) the management of data that are critical to the application itself and (2) the creation of an infrastructure for storage and retrieval of objects. In general, data management is designed in a layered fashion. The idea is to isolate the low-level requirements for manipulating data structures from the higher-level requirements for handling system attributes.

Within the system context, a database management system is often used as a common data store for all subsystems. The objects required to manipulate the database are members of reusable classes that are identified using domain analysis (Chapter 21) or are supplied directly by the database vendor. A detailed discussion of database design for OO systems is beyond the scope of this book.[9]

The design of the data management component includes the design of the attributes and operations required to manage objects. The relevant attributes are appended to every object in the problem domain and provide information that answers the question, "How do I store myself?" Coad and Yourdon [COA91] suggest the creation of an object-server class "with services to (a) tell each object to save itself and (b) retrieve stored objects for use by other design components."

As an example of data management for the **sensor** object discussed as part of the *SafeHome* security system, the design could specify a flat file called "sensor." Each record would correspond to a named instance of **sensor** and would contain the values of each **sensor** attribute for that named instance. Operations within the object-server class would enable a specific object to be stored and retrieved when it is needed

9 Interested readers should refer to [BRO91], [TAY92], or [RAO94].

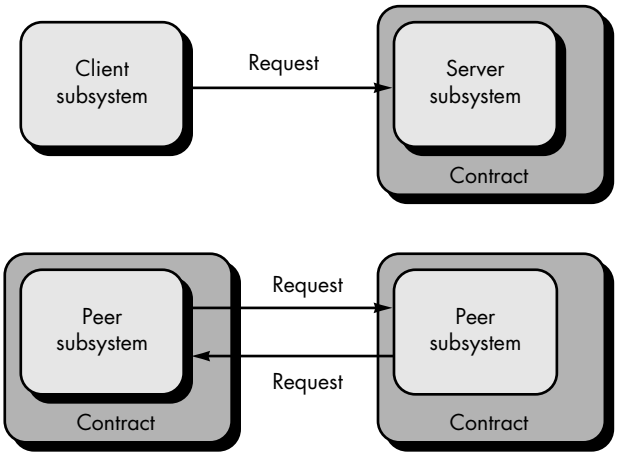


FIGURE 22.4 A model of collaboration between subsystems

by the system. For more complex objects, it might be necessary to specify a relational database or an object-oriented database to accomplish the same function.

22.2.6 The Resource Management Component

A variety of different resources are available to an OO system or product; and in many instances, subsystems compete for these resources at the same time. Global system resources can be external entities (e.g., a disk drive, processor, or communication line) or abstractions (e.g., a database, an object). Regardless of the nature of the resource, the software engineer should design a control mechanism for it. Rumbaugh and his colleagues [RUM91] suggest that each resource should be owned by a "guardian object." The guardian object is the gatekeeper for the resource, controlling access to it and moderating conflicting requests for it.

22.2.7 Intersubsystem Communication

Once each subsystem has been specified, it is necessary to define the collaborations that exist between the subsystems. The model that we use for object-to-object collaboration can be extended to subsystems as a whole. Figure 22.4 illustrates a collaboration model. As we noted earlier in this chapter, communication can occur by establishing a client/server link or a peer-to-peer link. Referring to the figure, we must specify the contract that exists between subsystems. Recall that a contract provides an indication of the ways in which one subsystem can interact with another.

? What design steps are required to specify a "contract" for a subsystem?

The following design steps can be applied to specify a contract for a subsystem [WIR90]:

1. **List each request that can be made by collaborators of the subsystem.** Organize the requests by subsystem and define them within one or more appropriate contracts. Be sure to note contracts that are inherited from superclasses.

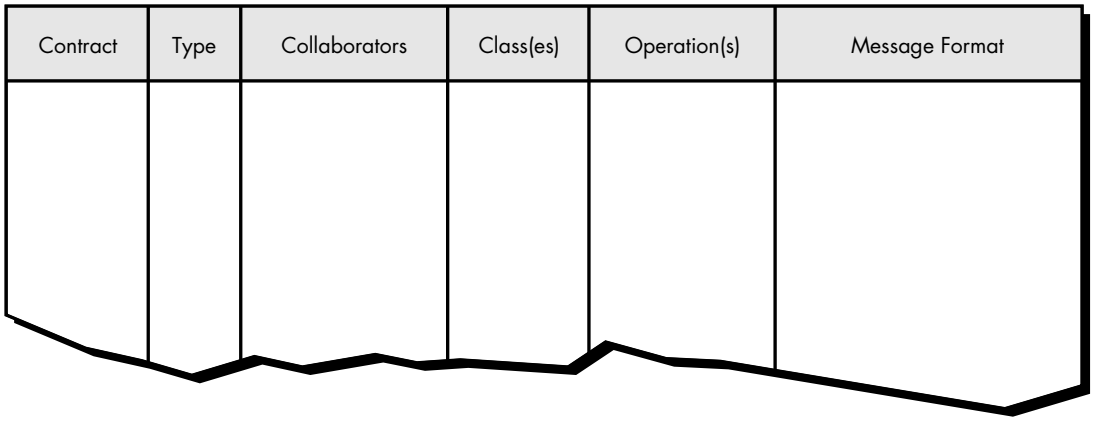


FIGURE 22.5 Subsystem collaboration table

2. **For each contract, note the operations (both inherited and private) that are required to implement the responsibilities implied by the contract.** Be sure to associate the operations with specific classes that reside within a subsystem.

Every contract between subsystems is manifested by one or more messages that move between objects within the subsystems.

3. **Considering one contract at a time, create a table of the form shown in Figure 22.5.** For each contract, the following entries are made in the table:

   Type—the type of contract (i.e., client/server or peer-to-peer).

   Collaborators—the names of the subsystems that are parties to the contract.

   Class—the names of the classes (contained within a subsystem) that support services implied by the contract.

   Operation—the names of the operations (within the class) that implement the services.

   Message format—the message format required to implement the interaction between collaborators.

   Draft an appropriate message description for each interaction between the subsystems.

4. **If the modes of interaction between subsystems are complex, a subsystem-collaboration diagram, illustrated in Figure 22.6 is created.** The collaboration graph is similar in form to the event flow diagram discussed in Chapter 21. Each subsystem is represented along with its interactions with other subsystems. The contracts that are invoked during an interaction are noted as shown. The details of the interaction are determined by looking up the contract in the subsystem collaboration table (Figure 22.5)

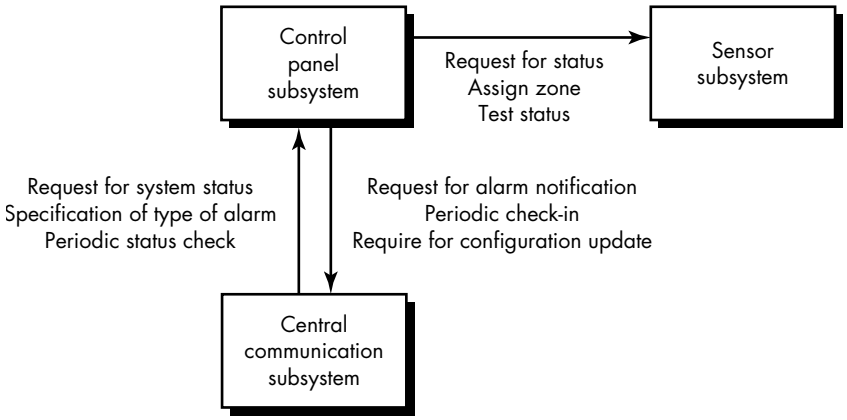


FIGURE 22.6 Abbreviated subsystem collaboration graph for SafeHome

22.3 THE OBJECT DESIGN PROCESS

Borrowing from a metaphor that was introduced earlier in this book, the OO system design might be viewed as the floor plan of a house. The floor plan specifies the purpose of each room and the architectural features that connect the rooms to one another and to the outside environment. It is now time to provide the details that are required to build each room. In the context of OOD, object design focuses on the "rooms."

Bennett and his colleagues [BEN99] discuss object design in the following way:

> Object design is concerned with the detailed design of the objects and their interactions. It is completed within the overall architecture defined during system design and according to agreed design guidelines and protocols. Object design is particularly concerned with the specification of attribute types, how operations function, and how objects are linked to other objects.

It is at this stage that the basic concepts and principles associated with component-level design (Chapter 16) come into play. Local data structures are defined (for attributes) and algorithms (for operations) are designed.

22.3.1 Object Descriptions

Be sure that the architecture has been defined before you begin working on object design. Don't let the architecture just happen.

A design description of an object (an instance of a class or subclass) can take one of two forms [GOL83]: (1) a *protocol description* that establishes the interface of an object by defining each message that the object can receive and the related operation that the object performs when it receives the message or (2) an *implementation description* that shows implementation details for each operation implied by a message that is passed to an object. Implementation details include information about the object's private part; that is, internal details about the data structures that describe the object's attributes and procedural details that describe operations.

The protocol description is nothing more than a set of messages and a corresponding comment for each message. For example, a portion of the protocol description for the object **motion sensor** (described earlier) might be

```
MESSAGE (motion.sensor) --> read: RETURNS sensor.ID, sensor.status;
```

This describes the message required to read the sensor. Similarly,

```
MESSAGE (motion.sensor) --> set: SENDS sensor.ID, sensor.status;
```

sets or resets the status of the sensor.

For a large system with many messages, it is often possible to create message categories. For example, message categories for the *SafeHome* **system** object might include system configuration messages, monitoring messages, event messages, and so forth.

To achieve the benefits of information hiding (Chapter 13), anyone who intends to use an object needs only the protocol description. The implementation description contains detail that should be "hidden" from those with no need to know.

An implementation description of an object provides the internal ("hidden") details that are required for implementation but are not necessary for invocation. That is, the designer of the object must provide an implementation description and must therefore create the internal details of the object. However, another designer or implementer who uses the object or other instances of the object requires only the protocol description but not the implementation description.

An implementation description is composed of the following information: (1) a specification of the object's name and reference to class; (2) a specification of private data structure with indication of data items and types; (3) a procedural description of each operation or, alternatively, pointers to such procedural descriptions. The implementation description must contain sufficient information to provide for proper handling of all messages described in the protocol description.

Cox [COX85] characterizes the difference between the information contained in the protocol description and that contained in the implementation description in terms of "users" and "suppliers" of services. A user of the service provided by an object must be familiar with the protocol for invoking the service; that is, for specifying what is desired. The supplier of the service (the object itself) must be concerned with how the service is to be supplied to the user; that is, with implementation details.

22.3.2 Designing Algorithms and Data Structures

A variety of representations contained in the analysis model and the system design provide a specification for all operations and attributes. Algorithms and data structures are designed using an approach that differs little from the data design and component-level design approaches discussed for conventional software engineering.

XRef

Virtually every concept presented in Chapter 13 is applicable here. Be sure you're familiar with the topics presented there.

An algorithm is created to implement the specification for each operation. In many cases, the algorithm is a simple computational or procedural sequence that can be implemented as a self-contained software module. However, if the specification of the operation is complex, it may be necessary to modularize the operation. Conventional component-level design techniques can be used to accomplish this.

Data structures are designed concurrently with algorithms. Since operations invariably manipulate the attributes of a class, the design of the data structures that best reflect the attributes will have a strong bearing on the algorithmic design of the corresponding operations.

? Is there a way to categorize operations (methods)?

Although many different types of operations exist, they can generally be divided into three broad categories: (1) operations that manipulate data in some way (e.g., adding, deleting, reformatting, selecting), (2) operations that perform a computation, and (3) operations that monitor an object for the occurrence of a controlling event.

For example, the *SafeHome* processing narrative contains the sentence fragments: "sensor is assigned a number and type" and "a master password is programmed for arming and disarming the system." These two phrases indicate a number of things:

- That an *assign* operation is relevant for the **sensor** object.
- That a *program* operation will be applied to the **system** object.
- That *arm* and *disarm* are operations that apply to **system** (also that **system status** may ultimately be defined (using data dictionary notation) as

```
system status = [armed | disarmed]
```

An operation is refined in much the same way that we refine a function in conventional design. Write a processing narrative, do a grammatical parse, and isolate new operations at a lower level of abstraction.

The operation *program* is allocated during OOA, but during object design it will be refined into a number of more specific operations that are required to configure the system. For example, after discussions with product engineering, the analyst, and possibly the marketing department, the designer might elaborate the original processing narrative and write the following for *program* (potential operations—verbs—are underlined):

Program enables the *SafeHome* user to configure the system once it has been installed. The user can (1) install phone numbers; (2) define delay times for alarms; (3) build a sensor table that contains each sensor ID, its type, and location; and (4) load a master password.

Therefore, the designer has refined the single operation *program* and replaced it with the operations: *install, define, build,* and *load*. Each of these new operations becomes part of the **system** object, has knowledge of the internal data structures that implement the object's attributes, and is invoked by sending the object messages of the form

```
MESSAGE (system) --> install: SENDS telephone.number;
```

This implies that, to provide the system with an emergency phone number, an *install* message will be sent to system.

Verbs connote actions or occurrences. In the context of object design formalization, we consider not only verbs but also descriptive verb phrases and predicates (e.g., "is equal to") as potential operations. The grammatical parse is applied recursively until each operation has been refined to its most-detailed level.

Once the basic object model is created, optimization should occur. Rumbaugh and his colleagues [RUM91] suggest three major thrusts for OOD design optimization:

- Review the object-relationship model to ensure that the implemented design leads to efficient utilization of resources and ease of implementation. Add redundancy where necessary.
- Revise attribute data structures and corresponding operation algorithms to enhance efficient processing.
- Create new attributes to save derived information, thereby avoiding recomputation.

A detailed discussion of OO design optimization is beyond the scope of this book. The interested reader should refer to [RUM91] and [CHA93]. For a discussion of how these concepts translate into the UML process, the reader should examine [JAC99] and [RUM99].

22.3.3 Program Components and Interfaces

An important aspect of software design quality is modularity; that is, the specification of program components (modules) that are combined to form a complete program. The object-oriented approach defines the object as a program component that is itself linked to other components (e.g., private data, operations). But defining objects and operations is not enough. During design, we must also identify the interfaces between objects and the overall structure (considered in an architectural sense) of the objects.

Although a program component is a design abstraction, it should be represented in the context of the programming language used for implementation. To accommodate OOD, the programming language to be used for implementation should be capable of creating the following program component (modeled after Ada):

```
PACKAGE  program-component-name IS
   TYPE specification of data objects
            •
            •
            •
   PROC specification of related operations . . .
PRIVATE
   data structure details for objects
PACKAGE BODY program-component-name IS
   PROC operation.1 (interface description) IS
            •
            •
            •
   END
   PROC operation.n (interface description) IS
            •
            •
            •
   END
END program-component-name
```

Referring to the Ada-like PDL (program design language) just shown, a program component is specified by indicating both data objects and operations. The *specification part* of the component indicates all data objects (declared with the **TYPE** statement) and the operations (**PROC** for procedure) that act on them. The *private part* (**PRIVATE**) of the component provides otherwise hidden details of data structure and processing. In the context of our earlier discussion, the **PACKAGE** is conceptually similar to objects discussed throughout this chapter.

The first program component to be identified should be the highest-level module from which all processing originates and all data structures evolve. Referring once again to the *SafeHome* example, we can define the highest-level program component as

```
PROCEDURE SafeHome software
```

The *SafeHome* software component can be coupled with a preliminary design for the following packages (objects):

```
PACKAGE system IS
   TYPE system data
   PROC install, define, build, load
   PROC display, reset, query, modify, call
   PRIVATE
      PACKAGE BODY system IS
      PRIVATE
            system.id IS STRING LENGTH (8);
            verification phone.number, telephone.number, ...
            IS STRING LENGTH (8);
            sensor.table DEFINED
                  sensor.type IS STRING LENGTH (2),
                  sensor.number, alarm.threshold IS NUMERIC;
      PROC install RECEIVES (telephone.number)
            {design detail for operation install}
      .
      .
      .
END system
PACKAGE sensor IS
   TYPE sensor data
   PROC read, set, test
   PRIVATE
      PACKAGE BODY sensor IS
      PRIVATE
            sensor.id IS STRING LENGTH (8);
            sensor.status IS STRING LENGTH (8);
            alarm.characteristics DEFINED
                  threshold, signal type, signal level IS NUMERIC,
            hardware.interface DEFINED
                  type, a/d.characteristics, timing.data IS NUMERIC,
```

```
END sensor
   .
   .
   .
END SafeHome software
```

Data objects and corresponding operations are specified for each of the program components for *SafeHome* software. The final step in the object design process completes all information required to fully implement data structure and types contained in the **PRIVATE** portion of the package and all procedural detail contained in the **PACKAGE BODY**.

To illustrate the detail design of a program component, we reconsider the **sensor** package described earlier. The data structures for **sensor** attributes have already been defined. Therefore, the first step is to define the interfaces for each of the operations attached to **sensor:**

```
PROC read (sensor.id, sensor.status: OUT);
PROC set (alarm.characteristics, hardware.interface: IN)
PROC test (sensor.id, sensor.status, alarm.characteristics: OUT);
```

Stepwise refinement and structured programming (Chapter 16) are used at this stage to complete the design of each operation.

The next step requires stepwise refinement of each operation associated with the **sensor** package. To illustrate the refinement, we develop a processing narrative (an informal strategy) for *read:*

When the sensor object receives a *read* message, the *read* process is invoked. The process determines the interface and signal type, polls the sensor interface, converts A/D characteristics into an internal signal level, and compares the internal signal level to a threshold value. If the threshold is exceeded, the sensor status is set to "event." Otherwise, the sensor status is set to "no event." If an error is sensed while polling the sensor, the sensor status is set to "error."

Given the processing narrative, a PDL description of the *read* process can be developed:

```
PROC read (sensor.id, sensor.status: OUT);
   raw.signal IS BIT STRING
   IF (hardware.interface.type = "s" & alarm.characteristics.signal.type = "B"
   THEN
      GET (sensor, exception: sensor.status := error) raw.signal;
      CONVERT raw.signal TO internal.signal.level;
      IF internal.signal.level > threshold
         THEN sensor.status := "event";
         ELSE sensor.status := "no event";
      ENDIF
   ELSE {processing for other types of s interfaces would be specified}
   ENDIF
   RETURN sensor.id, sensor.status;
END read
```

The PDL representation of the *read* operation can be translated into the appropriate implementation language. The functions **GET** and **CONVERT** are assumed to be available as part of a run-time library.

22.4 DESIGN PATTERNS

The best designers in any field have an uncanny ability to see patterns that characterize a problem and corresponding patterns that can be combined to create a solution. Gamma and his colleagues [GAM95] discuss this when they state:

XRef

Patterns exist at the architecture and the component levels. For further discussion, see Chapter 14.

> [Y]ou'll find recurring patterns of classes and communicating objects in many object-oriented systems. These patterns solve specific design problems and make object-oriented design more flexible, elegant, and ultimately reusable. They help designers reuse successful designs by basing new designs on prior experience. A designer who is familiar with such patterns can apply them immediately to design problems without having to rediscover them.

Throughout the OOD process, a software engineer should look for every opportunity to reuse existing design patterns (when they meet the needs of the design) rather than creating new ones.

WebRef

An excellent paper entitled "Non-Software Examples of Software Design Patterns" provides insight:
www.agcs.com/patterns/papers/patexamples.htm

22.4.1 Describing a Design Pattern

Mature engineering disciplines make use of thousands of design patterns. For example, a mechanical engineer uses a two-step, keyed shaft as a design pattern. Inherent in the pattern are attributes (the diameters of the shaft, the dimensions of the keyway, etc.) and operations (e.g., shaft rotation, shaft connection). An electrical engineer uses an integrated circuit (an extremely complex design pattern) to solve a specific element of a new problem. All design patterns can be described by specifying the following information [GAM95]:

Quote:

"[Patterns] constitute a 'grass roots' effort to build on the collective experience of skilled designers and software engineers."
Frank Buschmann et al.

- the name of the pattern
- the intent of the pattern
- the "design forces" that motivate the pattern
- the solution that mitigates these forces
- the classes that are required to implement the solution
- the responsibilities and collaboration among solution classes
- guidance that leads to effective implementation
- example source code or source code templates
- cross-references to related design patterns

The design pattern name is itself an abstraction that conveys significant meaning once the applicability and intent are understood. *Design forces* describe the data, functional, or behavioral requirements associated with part of the software for which the

pattern is to be applied. In addition forces define the constraints that may restrict the manner in which the design is to be derived. In essence, design forces describe the environment and conditions that must exist to make the design pattern applicable. The pattern characteristics (classes, responsibilities, and collaborations) indicate the attributes of the design that may be adjusted to enable the pattern to accommodate a variety of problems. These attributes represent characteristics of the design that can be searched (e.g., via a database) so that an appropriate pattern can be found. Finally, guidance associated with the use of a design pattern provides an indication of the ramifications of design decisions.

Quote:
"A design pattern becomes an AntiPattern when it creates more problems than it solves."
William Brown et al.

The names of objects and subsystems (potential design patterns) should be chosen with care. As we discuss in Chapter 27, one of the key technical problems in software reuse is simply the inability to find existing reusable patterns when hundreds or thousands of candidate patterns exist. The search for the "right" pattern is aided immeasurably by a meaningful pattern name along with a set of characteristics that help in classifying the object [PRE95].

22.4.2 Using Patterns in Design

WebRef
The Portland Pattern Repository publishes an evolving collection of design patterns at:
c2.com/ppr

In an object-oriented system, design patterns[10] can be used by applying two different mechanisms: inheritance and composition. Inheritance is a fundamental OO concept and was described in detail in Chapter 20. Using inheritance, an existing design pattern becomes a template for a new subclass. The attributes and operations that exist in the pattern become part of the subclass.

Composition is a concept that leads to aggregate objects. That is, a problem may require objects that have complex functionality (in the extreme, a subsystem accomplishes this). The complex object can be assembled by selecting a set of design patterns and composing the appropriate object (or subsystem). Each design pattern is treated as a black box, and communication among the patterns occurs only via well-defined interfaces.

ADVICE
Good design always strives for simplicity. Therefore, opt for composition when it leads to simpler inheritance structures.

Gamma and his colleagues [GAM95] suggest that object composition should be favored over inheritance when both options exist. Rather than creating large and sometimes unmanageable class hierarchies (the consequence of the overuse of inheritance), composition favors small class hierarchies and objects that remain focused on one objective. Composition uses existing design patterns (reusable components) in an unaltered form.

22.5 OBJECT-ORIENTED PROGRAMMING

Although all areas of object technologies have received significant attention within the software community, no subject has produced more books, more discussion, and

10 Buschmann [BUS96] and Gamma et al. [GAM95] among many others have written catalogs of design patterns for use in OO systems.

more debate than *object-oriented programming* (OOP). Hundreds of books have been written on C++ and Java programming, and hundreds more are dedicated to less widely used OO languages.

The software engineering viewpoint stresses OOA and OOD and considers OOP (coding) an important, but secondary, activity that is an outgrowth of analysis and design. The reason for this is simple. As the complexity of systems increases, the design architecture of the end product has a significantly stronger influence on its success than the programming language that has been used. And yet, "language wars" continue to rage.

The details of OOP are best left to books dedicated to the subject. The interested reader should refer to one or more of the OOP books noted in the Further Readings and Information Sources section at the end of this chapter.

22.6 SUMMARY

Object-oriented design translates the OOA model of the real world into an implementation-specific model that can be realized in software. The OOD process can be described as a pyramid composed of four layers. The foundation layer focuses on the design of subsystems that implement major system functions. The class layer specifies the overall object architecture and the hierarchy of classes required to implement a system. The message layer indicates how collaboration between objects will be realized, and the responsibilities layer identifies the attributes and operations that characterize each class.

Like OOA, there are many different OOD methods. UML is an attempt to provide a single approach to OOD that is applicable in all application domains. UML and other methods approach the design process through two levels of abstraction—design of subsystems (architecture) and design of individual objects.

During system design, the architecture of the object-oriented system is developed. In addition to developing subsystems, their interactions, and their placement in architectural layers, system design considers the user interaction component, a task management component, and a data management component. These subsystem components provide a design infrastructure that enables the application to operate effectively. The object design process focuses on the description of data structures that implement class attributes, algorithms that implement operations, and messages that enable collaborations and object relationships.

Design patterns allow the designer to create the system architecture by integrating reusable components. Object-oriented programming extends the design model into the executable domain. An OO programming language is used to translate the classes, attributes, operations, and messages into a form that can be executed by a machine.

REFERENCES

[BEN99] Bennett, S., S. McRobb, and R. Farmer, *Object Oriented System Analysis and Design Using UML,* McGraw-Hill, 1999.

[BIH92] Bihari, T. and P. Gopinath, "Object-Oriented Real-Time Systems: Concepts and Examples," *Computer,* vol. 25, no. 12, December 1992, pp. 25–32.

[BOO94] Booch, G., *Object-Oriented Analysis and Design,* 2nd ed., Benjamin Cummings, 1994.

[BOO99] Booch, G., I. Jacobson, J. Rumbaugh, *The Unified Modeling Language User Guide,* Addison-Wesley, 1999.

[BRO91] Brown, A.W., *Object-Oriented Databases,* McGraw-Hill, 1991.

[BUS96] Buschmann, F., et al., *A System of Patterns: Pattern Oriented System Architecture,* Wiley, 1996.

[CHA93] De Champeaux, D., D. Lea, and P. Faure, *Object-Oriented System Development,* Addison-Wesley, 1993.

[COA91] Coad, P. and E. Yourdon, *Object-Oriented Design,* Prentice-Hall, 1991.

[COX85] Cox, B., "Software ICs and Objective-C," *UnixWorld,* Spring 1985.

[DAV95] Davis, A., "Object-Oriented Requirements to Object-Oriented Design: An Easy Transition?" *Journal of Systems Software,* vol. 30, 1995, pp. 151–159.

[DOU99] Douglass, B., *Real-Time UML: Developing Efficient Objects for Embedded Systems,* Addison-Wesley, 1999.

[FIC92] Fichman, R. and C. Kemerer, "Object-Oriented and Conceptual Design Methodologies," *Computer,* vol. 25, no. 10, October 1992, pp. 22–39.

[GAM95] Gamma, E., et al., *Design Patterns,* Addison-Wesley, 1995.

[GOL83] Goldberg, A. and D. Robson, *Smalltalk-80: The Language and Its Implementation,* Addison-Wesley, 1983.

[JAC92] Jacobson, I., *Object-Oriented Software Engineering,* Addison-Wesley, 1992.

[JAC99] Jacobson, I., G. Booch, J. Rumbaugh, *Unified Software Development Process,* Addison-Wesley, 1999.

[MEY90] Meyer, B., *Object-Oriented Software Construction,* 2nd ed., Prentice-Hall, 1988.

[PRE95] Pree, W., *Design Patterns for Object-Oriented Software Development,* Addison-Wesley, 1995.

[RUM91] Rumbaugh, J., et al., *Object-Oriented Modeling and Design,* Prentice-Hall, 1991.

[RUM99] Rumbaugh, J., I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual,* Addison-Wesley, 1999.

[RAO94] Rao, B.A., *Object-Oriented Databases: Technology, Applications and Products,* McGraw-Hill, 1994.

[TAY92} Taylor, D.A., *Object-Oriented Information Systems,* Wiley, 1992.

[WIR90] Wirfs-Brock, R., B. Wilkerson, and L. Weiner, *Designing Object-Oriented Software,* Prentice-Hall, 1990.

PROBLEMS AND POINTS TO PONDER

22.1. The design pyramid for OOD differs somewhat from the pyramid described for conventional software design (Chapter 13). Discuss the differences and similarities of the two pyramids.

22.2. How do OOD and structured design differ? What aspects of these two design methods are the same?

22.3. Review the five criteria for effective OO modularity discussed in Section 22.1.2. Using the design approach described later in the chapter, demonstrate how these five criteria are achieved.

22.4. Using outside references on UML, prepare a one-hour tutorial for your class. Be sure to show all important diagrammatic modeling conventions used in UML.

22.5. Select an older OOD method presented in Section 22.1.3 and prepare a one-hour tutorial for your class. Be sure to show all important diagrammatic modeling conventions that the authors suggest.

22.6. Discuss how the use-case can serve as an important source of information for design.

22.7. Research a GUI development environment and show how the user interaction component is implemented in the real world. What design patterns are offered and how are they used?

22.8. Task management for OO systems can be quite complex. Do some research of OOD methods for real-time systems (e.g., [BIH92] or [DOU99]) and determine how task management is achieved in that context.

22.9. Discuss how the data management component is implemented in a typical OO development environment.

22.10. Write a two- or three-page paper on object-oriented databases and discuss how they might be used to develop the data management component.

22.11. How does a designer recognize tasks that must be concurrent?

22.12. Apply the OOD approach discussed in this chapter to flesh out the design for the *SafeHome* system. Define all relevant subsystems and develop object designs for important classes.

22.13. Apply OOD approach discussed in this chapter to the PHTRS system described in Problem 12.13.

22.14. Describe a video game and apply OOD approach discussed in this chapter to represent its design.

22.15. You are responsible for the development of an electronic mail (e-mail) system to be implemented on a PC network. The e-mail system will enable users to create letters to be mailed to another user, general distribution, or a specific address list. Letters can be read, copied, stored, and the like. The e-mail system will use existing word-processing capability to create letters. Using this description as a starting point, derive a set of requirements and apply OOD techniques to create a top-level design for the e-mail system.

22.16. A small island nation has decided to build an air traffic control (ATC) system for its one airport. The system is specified as follows:

> All aircraft landing at the airport must have a transponder that transmits aircraft type and flight data in high-density packed format to the ATC ground station. The ATC ground station can query an aircraft for specific information. When the ATC ground station receives data, it is unpacked and stored in an aircraft database. A computer graphics display is created from the stored information and displayed for an air traffic controller. The display is updated every 2 seconds. All information is analyzed to determine if "dangerous situations" are present. The air traffic controller can query the database for specific information about any plane displayed on the screen.

Using OOD, create a design for the ATC system. Do not attempt to implement it!

FURTHER READINGS AND INFORMATION SOURCES

In addition to the many references in this chapter, books by Gossain and Graham (*Object Modeling and Design Strategies,* SIGS Books, 1998); Meyer (*Object-Oriented Software Construction,* 2nd ed., Prentice-Hall, 1997); Reil (*Object-Oriented Design Through Heuristics,* Addison-Wesley, 1996); and Walden and Nerson (*Seamless Object-Oriented Software Architecture: Analysis and Design of Reliable Systems,* Prentice-Hall, 1995) cover OOD in considerable detail. Fowler (*Refactoring: Improving the Design of Existing Code,* Addison-Wesley, 1999) addresses the use of object-oriented techniques to redesign and rebuild old programs to improve their design quality.

Many recent books published on object-oriented design emphasize UML. Those serious about applying UML in their work should acquire [BOO99], [RUM99], and [JAC99]. In addition, many of the books referenced in the Further Reading and Information Sources section of Chapter 21 also address design in considerable detail.

The use of design patterns for the development of object-oriented software has important implications for component-based software engineering, reusability in general, and the overall quality of resultant systems. In addition to [BUS96] and [GAM95], many recent books are dedicated to the subject:

Ambler, S.W., *Process Patterns: Building Large-Scale Systems Using Object Technology,* Cambridge University Press, 1999.

Coplien, J.O. and D.C. Schmidt, *Pattern Languages of Program Design,* Addison-Wesley, 1995.

Fowler, M., *Analysis Patterns: Reusable Object Models,* Addison-Wesley, 1996.

Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design,* Prentice-Hall, 1997.

Martin, R.C., et al., *Pattern Languages of Program Design 3,* Addison-Wesley, 1997.

Rising, L. and J. Coplien (eds.), *The Patterns Handbook: Techniques, Strategies, and Applications,* SIGS Books, 1998.

Pree, W., *Design Patterns for Object-Oriented Software Development,* Addison-Wesley, 1995.

Vlissides, J., Pattern Hatching: *Design Patterns Applied,* Addison-Wesley, 1998.

Vlissides, J.M., J.O. Coplien, and N. Kerth, *Pattern Languages of Program Design 2,* Addison-Wesley, 1996.

Hundreds of books have been published on object-oriented programming. A sampling of OOP language-specific books follows:

C++: Cohoon, J.P., *C++ Program Design: An Introduction to Programming and Object-Oriented Design,* McGraw Hill, 1998.

Barclay, K. and J. Savage, *Object-Oriented Design with C++,* Prentice-Hall, 1997.

Eiffel: Thomas, P. and R. Weedon, *Object-Oriented Programming in Eiffel,* Addison-Wesley, 1997.

Jezequel, J.M., *Object-Oriented Software Engineering with Eiffel,* Addison-Wesley, 1996.

Java: Coad, P., M. Mayfield, and J. Kern, *Java Design: Building Better Apps and Applets,* 2nd ed., Prentice-Hall, 1998.

Lewis, J. and W. Loftus, *Java Software Solutions: Foundations of Program,* Addison-Wesley, 1997.

Smalltalk: Sharp, A., *Smalltalk by Example: The Developer's Guide,* McGraw-Hill, 1997.

LaLonde, W.R. and J.R. Pugh, *Programming in Smalltalk,* Prentice-Hall, 1995.

Books that cover OOD topics using two or more OO programming languages provide insight and comparison of language features. Titles include:

Drake, C., *Object-Oriented Programming With C++ and Smalltalk,* Prentice-Hall, 1998.

Joyner, I., *Objects Unencapsulated: Java, Eiffel and C++,* Prentice-Hall, 1999.

Zeigler, B.P., *Objects and Systems: Principled Design with Implementations in C++ and Java,* Springer-Verlag, 1997.

A wide variety of information sources on object-oriented design and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to OOD can be found at the SEPA Web site:

http://www.mhhe.com/engcs/compsci/pressman/resources/OOD.mhtml

CHAPTER

23 OBJECT-ORIENTED TESTING

KEY CONCEPTS

The objective of testing, stated simply, is to find the greatest possible number of errors with a manageable amount of effort applied over a realistic time span. Although this fundamental objective remains unchanged for object-oriented software, the nature of OO programs changes both testing strategy and testing tactics.

It might be argued that, as OOA and OOD mature, greater reuse of design patterns will mitigate the need for heavy testing of OO systems. Exactly the opposite is true. Binder [BIN94b] discusses this when he states:

> [E]ach reuse is a new context of usage and retesting is prudent. It seems likely that more, not less, testing will be needed to obtain high reliability in object-oriented systems.

The testing of OO systems presents a new set of challenges to the software engineer. The definition of testing must be broadened to include error discovery techniques (formal technical reviews) applied to OOA and OOD models. The completeness and consistency of OO representations must be assessed as they are built. Unit testing loses much of its meaning, and integration strategies change significantly. In summary, both testing strategies and testing tactics must account for the unique characteristics of OO software.

QUICK LOOK

What is it? The architecture of object-oriented software results in a series of layered subsystems that encapsulate collaborating classes. Each of these system elements (subsystems and classes) perform functions that help to achieve system requirements. It is necessary to test an OO system at a variety of different levels in an effort to uncover errors that may occur as classes collaborate with one another and subsystems communicate across architectural layers.

Who does it? Object-oriented testing is performed by software engineers and testing specialists.

Why is it important? You have to execute the program before it gets to the customer with the specific intent of removing all errors, so that the customer will not experience the frustration associated with a poor-quality product. In order to find the highest possible number of errors, tests must be conducted systematically and test cases must be designed using disciplined techniques.

What are the steps? OO testing is strategically similar to the testing of conventional systems, but it is tactically different. Because the OO analysis and design models are similar in structure and content to the resultant OO program, "testing" begins with the review of these models. Once code has been generated, OO testing begins "in the small" with class testing. A series of tests are designed that exercise class operations and examine

QUICK LOOK

whether errors exist as one class collaborates with other classes. As classes are integrated to form a subsystem, thread-based, use-based, and cluster testing, along with fault-based approaches, are applied to fully exercise collaborating classes. Finally, use-cases (developed as part of the OO analysis model) are used to uncover errors at the software validation level.

What is the work product? A set of test cases to exercise classes, their collaborations, and behaviors is designed and documented; expected results defined; and actual results recorded.

How do I ensure that I've done it right? When you begin testing, change your point of view. Try hard to "break" the software! Design test cases in a disciplined fashion and review the test cases you do create for thoroughness.

23.1 BROADENING THE VIEW OF TESTING

The construction of object-oriented software begins with the creation of analysis and design models (Chapters 21 and 22). Because of the evolutionary nature of the OO software engineering paradigm, these models begin as relatively informal representations of system requirements and evolve into detailed models of classes, class connections and relationships, system design and allocation, and object design (incorporating a model of object connectivity via messaging). At each stage, the models can be tested in an attempt to uncover errors prior to their propagation to the next iteration.

Quote:

"Because of their ability to detect and correct defects in upstream work products, technical reviews are at least as important in controlling cost and schedule as testing."

Steve McConnell

It can be argued that the review of OO analysis and design models is especially useful because the same semantic constructs (e.g., classes, attributes, operations, messages) appear at the analysis, design, and code levels. Therefore, a problem in the definition of class attributes that is uncovered during analysis will circumvent side effects that might occur if the problem were not discovered until design or code (or even the next iteration of analysis).

For example, consider a class in which a number of attributes are defined during the first iteration of OOA. An extraneous attribute is appended to the class (due to a misunderstanding of the problem domain). Two operations are then specified to manipulate the attribute. A review is conducted and a domain expert points out the error. By eliminating the extraneous attribute at this stage, the following problems and unnecessary effort may be avoided during analysis:

1. Special subclasses may have been generated to accommodate the unnecessary attribute or exceptions to it. Work involved in the creation of unnecessary subclasses has been avoided.
2. A misinterpretation of the class definition may lead to incorrect or extraneous class relationships.

3. The behavior of the system or its classes may be improperly characterized to accommodate the extraneous attribute.

If the error is not uncovered during analysis and propagated further, the following problems could occur (and will have been avoided because of the earlier review) during design:

1. Improper allocation of the class to subsystem and/or tasks may occur during system design.
2. Unnecessary design work may be expended to create the procedural design for the operations that address the extraneous attribute.
3. The messaging model will be incorrect (because messages must be designed for the operations that are extraneous).

There's an old saying about "nipping problems in the bud." If you spend time reviewing the OOA and OOD models, that's what you'll do.

If the error remains undetected during design and passes into the coding activity, considerable effort will be expended to generate code that implements an unnecessary attribute, two unnecessary operations, messages that drive interobject communication, and many other related issues. In addition, testing of the class will absorb more time than necessary. Once the problem is finally uncovered, modification of the system must be carried out with the ever-present potential for side effects that are caused by change.

During later stages of their development, OOA and OOD models provide substantial information about the structure and behavior of the system. For this reason, these models should be subjected to rigorous review prior to the generation of code.

All object-oriented models should be tested (in this context, the term *testing* is used to incorporate formal technical reviews) for correctness, completeness, and consistency [MGR94] within the context of the model's syntax, semantics, and pragmatics [LIN94].

23.2 TESTING OOA AND OOD MODELS

Analysis and design models cannot be tested in the conventional sense, because they cannot be executed. However, formal technical reviews (Chapter 8) can be used to examine the correctness and consistency of both analysis and design models.

23.2.1 Correctness of OOA and OOD Models

The notation and syntax used to represent analysis and design models will be tied to the specific analysis and design method that is chosen for the project. Hence, syntactic correctness is judged on proper use of the symbology; each model is reviewed to ensure that proper modeling conventions have been maintained.

During analysis and design, semantic correctness must be judged based on the model's conformance to the real world problem domain. If the model accurately

reflects the real world (to a level of detail that is appropriate to the stage of development at which the model is reviewed), then it is semantically correct. To determine whether the model does, in fact, reflect the real world, it should be presented to problem domain experts, who will examine the class definitions and hierarchy for omissions and ambiguity. Class relationships (instance connections) are evaluated to determine whether they accurately reflect real world object connections.[1]

23.2.2 Consistency of OOA and OOD Models

The consistency of OOA and OOD models may be judged by "considering the relationships among entities in the model. An inconsistent model has representations in one part that are not correctly reflected in other portions of the model" [MGR94].

OOA Models

To assess consistency, each class and its connections to other classes should be examined. The class-responsibility-collaboration model and an object-relationship diagram can be used to facilitate this activity. As we noted in Chapter 21, the CRC model is composed on CRC index cards. Each CRC card lists the class name, its responsibilities (operations), and its collaborators (other classes to which it sends messages and on which it depends for the accomplishment of its responsibilities). The collaborations imply a series of relationships (i.e., connections) between classes of the OO system. The object-relationship model provides a graphic representation of the connections between classes. All of this information can be obtained from the OOA model (Chapter 21).

To evaluate the class model the following steps have been recommended [MGR94]:

What steps should we take to review the class model?

1. **Revisit the CRC model and the object-relationship model.** Cross check to ensure that all collaborations implied by the OOA model are properly represented.
2. **Inspect the description of each CRC index card to determine if a delegated responsibility is part of the collaborator's definition.** For example, consider a class defined for a point-of-sale checkout system, called *credit sale.* This class has a CRC index card illustrated in Figure 23.1. For this collection of classes and collaborations, we ask whether a responsibility (e.g., *read credit card*) is accomplished if delegated to the named collaborator (**credit card**). That is, does the class **credit card** have an operation that enables it to be read? In this case the answer is, "Yes." The object-relationship is traversed to ensure that all such connections are valid.
3. **Invert the connection to ensure that each collaborator that is asked for service is receiving requests from a reasonable source.** For example, if the **credit card** class receives a request for **purchase amount** from the **credit sale** class, there would be a problem. **Credit card** does not know the **purchase amount.**

XRef

Additional suggestions for conducting a CRC model review are presented in Chapter 21.

1 Use-cases can be invaluable in tracking analysis and design models against real world usage scenarios for the OO system.

Class name: Credit sale	
Class type: Transaction event	
Class characteristics: Nontangible, atomic, sequential, permanent, guarded	
Responsibilities:	Collaborators:
Read credit card	Credit card
Get authorization	Credit authority
Post purchase amount	Product ticket
	Sales ledger
	Audit file
Generate bill	Bill

FIGURE 23.1 An example CRC index card used for review

4. **Using the inverted connections examined in step 3, determine whether other classes might be required and whether responsibilities are properly grouped among the classes.**
5. **Determine whether widely requested responsibilities might be combined into a single responsibility.** For example, *read credit card* and *get authorization* occur in every situation. They might be combined into a *validate credit request* responsibility that incorporates getting the credit card number and gaining authorization.
6. **Steps 1 through 5 are applied iteratively to each class and through each evolution of the OOA model.**

OOD Model

Once the OOD model (Chapter 22) is created, reviews of the system design and the object design should also be conducted. The system design depicts the overall product architecture, the subsystems that compose the product, the manner in which subsystems are allocated to processors, the allocation of classes to subsystems, and the design of the user interface. The object model presents the details of each class and the messaging activities that are necessary to implement collaborations between classes.

The system design is reviewed by examining the object-behavior model developed during OOA and mapping required system behavior against the subsystems designed to accomplish this behavior. Concurrency and task allocation are also reviewed within the context of system behavior. The behavioral states of the system are evaluated to determine which exist concurrently. Use-case scenarios are used to exercise the user interface design.

The object model should be tested against the object-relationship network to ensure that all design objects contain the necessary attributes and operations to implement the collaborations defined for each CRC index card. In addition, the detailed specification of operation details (i.e., the algorithms that implement the operations) are reviewed using conventional inspection techniques.

23.3 OBJECT-ORIENTED TESTING STRATEGIES

Quote:

"The best tester isn't the one who finds the most bugs . . . The best tester is the one who gets the most bugs fixed."

Cem Kaner et al.

The classical strategy for testing computer software begins with "testing in the small" and works outward toward "testing in the large." Stated in the jargon of software testing (Chapter 18), we begin with unit testing, then progress toward integration testing, and culminate with validation and system testing. In conventional applications, unit testing focuses on the smallest compilable program unit—the subprogram (e.g., module, subroutine, procedure, component). Once each of these units has been tested individually, it is integrated into a program structure while a series of regression tests are run to uncover errors due to interfacing between the modules and side effects caused by the addition of new units. Finally, the system as a whole is tested to ensure that errors in requirements are uncovered.

23.3.1 Unit Testing in the OO Context

When object-oriented software is considered, the concept of the unit changes. Encapsulation drives the definition of classes and objects. This means that each class and each instance of a class (object) packages attributes (data) and the operations (also known as *methods* or *services*) that manipulate these data. Rather than testing an individual module, the smallest testable unit is the encapsulated class or object. Because a class can contain a number of different operations and a particular operation may exist as part of a number of different classes, the meaning of unit testing changes dramatically.

KEY POINT

Class testing for OO software is equivalent to module unit testing for conventional software. It is not advisable to test operations in isolation.

We can no longer test a single operation in isolation (the conventional view of unit testing) but rather as part of a class. To illustrate, consider a class hierarchy in which an operation *X* is defined for the superclass and is inherited by a number of subclasses. Each subclass uses operation *X,* but it is applied within the context of the private attributes and operations that have been defined for the subclass. Because the context in which operation *X* is used varies in subtle ways, it is necessary to test operation *X* in the context of each of the subclasses. This means that testing operation *X* in a vacuum (the traditional unit testing approach) is ineffective in the object-oriented context.

Class testing for OO software is the equivalent of unit testing for conventional software.[2] Unlike unit testing of conventional software, which tends to focus on the algorithmic detail of a module and the data that flow across the module interface, class

2 Test case design methods for OO classes are discussed in Sections 23.4 through 23.6.

testing for OO software is driven by the operations encapsulated by the class and the state behavior of the class.

23.3.2 Integration Testing in the OO Context

Because object-oriented software does not have a hierarchical control structure, conventional top-down and bottom-up integration strategies have little meaning. In addition, integrating operations one at a time into a class (the conventional incremental integration approach) is often impossible because of the "direct and indirect interactions of the components that make up the class" [BER93].

There are two different strategies for integration testing of OO systems [BIN94a]. The first, *thread-based testing,* integrates the set of classes required to respond to one input or event for the system. Each *thread* is integrated and tested individually. Regression testing is applied to ensure that no side effects occur. The second integration approach, *use-based testing,* begins the construction of the system by testing those classes (called *independent classes*) that use very few (if any) of server classes. After the independent classes are tested, the next layer of classes, called *dependent classes,* that use the independent classes are tested. This sequence of testing layers of dependent classes continues until the entire system is constructed. Unlike conventional integration, the use of drivers and stubs (Chapter 18) as replacement operations is to be avoided, when possible.

The OO testing integration strategy focuses on groups of classes that collaborate or communicate in some manner.

Cluster testing [MGR94] is one step in the integration testing of OO software. Here, a cluster of collaborating classes (determined by examining the CRC and object-relationship model) is exercised by designing test cases that attempt to uncover errors in the collaborations.

23.3.3 Validation Testing in an OO Context

At the validation or system level, the details of class connections disappear. Like conventional validation, the validation of OO software focuses on user-visible actions and user-recognizable output from the system. To assist in the derivation of validation tests, the tester should draw upon the use-cases (Chapter 20) that are part of the analysis model. The use-case provides a scenario that has a high likelihood of uncovered errors in user interaction requirements.

XRef

Virtually all of the black-box testing methods discussed in Chapter 17 are applicable for OO.

Conventional black-box testing methods can be used to drive validations tests. In addition, test cases may be derived from the object-behavior model and from event flow diagram created as part of OOA.

23.4 TEST CASE DESIGN FOR OO SOFTWARE

Test case design methods for OO software are still evolving. However, an overall approach to OO test case design has been defined by Berard [BER93]:

1. Each test case should be uniquely identified and explicitly associated with the class to be tested.

2. The purpose of the test should be stated.
3. A list of testing steps should be developed for each test and should contain [BER93]:
   a. A list of specified states for the object that is to be tested.
   b. A list of messages and operations that will be exercised as a consequence of the test.
   c. A list of exceptions that may occur as the object is tested.
   d. A list of external conditions (i.e., changes in the environment external to the software that must exist in order to properly conduct the test).
   e. Supplementary information that will aid in understanding or implementing the test.

Unlike conventional test case design, which is driven by an input-process-output view of software or the algorithmic detail of individual modules, object-oriented testing focuses on designing appropriate sequences of operations to exercise the states of a class.

23.4.1 The Test Case Design Implications of OO Concepts

As we have already seen, the OO class is the target for test case design. Because attributes and operations are encapsulated, testing operations outside of the class is generally unproductive. Although encapsulation is an essential design concept for OO, it can create a minor obstacle when testing. As Binder [BIN94a] notes, "Testing requires reporting on the concrete and abstract state of an object." Yet, encapsulation can make this information somewhat difficult to obtain. Unless built-in operations are provided to report the values for class attributes, a snapshot of the state of an object may be difficult to acquire.

An excellent collection of papers, resources, and a bibliography on OO testing can be found at **www.rbsc.com**

Inheritance also leads to additional challenges for the test case designer. We have already noted that each new context of usage requires retesting, even though reuse has been achieved. In addition, multiple inheritance[3] complicates testing further by increasing the number of contexts for which testing is required [BIN94a]. If subclasses instantiated from a superclass are used within the same problem domain, it is likely that the set of test cases derived for the superclass can be used when testing the subclass. However, if the subclass is used in an entirely different context, the superclass test cases will have little applicability and a new set of tests must be designed.

23.4.2 Applicability of Conventional Test Case Design Methods

The white-box testing methods described in Chapter 17 can be applied to the operations defined for a class. Basis path, loop testing, or data flow techniques can help to ensure that every statement in an operation has been tested. However, the con-

3 An OOD concept that should be used with extreme care.

cise structure of many class operations causes some to argue that the effort applied to white-box testing might be better redirected to tests at a class level.

Black-box testing methods are as appropriate for OO systems as they are for systems developed using conventional software engineering methods. As we noted earlier in this chapter, use-cases can provide useful input in the design of black-box and state-based tests [AMB95].

The strategy is to hypothesize a set of plausible faults and then derive tests to prove the hypothesis.

23.4.3 Fault-Based Testing[4]

The object of *fault-based testing* within an OO system is to design tests that have a high likelihood of uncovering plausible faults. Because the product or system must conform to customer requirements, the preliminary planning required to perform fault-based testing begins with the analysis model. The tester looks for plausible faults (i.e., aspects of the implementation of the system that may result in defects). To determine whether these faults exist, test cases are designed to exercise the design or code.

Consider a simple example.[5] Software engineers often make errors at the boundaries of a problem. For example, when testing a SQRT operation that returns errors for negative numbers, we know to try the boundaries: a negative number close to zero and zero itself. "Zero itself" checks whether the programmer made a mistake like

```
if (x > 0) calculate_the_square_root();
```

instead of the correct

```
if (x >= 0) calculate_the_square_root();
```

Because fault-based testing occurs at a detailed level, it is best reserved for operations and classes that are critical or suspect.

As another example, consider a Boolean expression:

```
if (a && !b || c)
```

Multicondition testing and related techniques probe for certain plausible faults in this expression, such as

&& should be **||**

! was left out where it was needed

There should be parentheses around **!b || c**

For each plausible fault, we design test cases that will force the incorrect expression to fail. In the previous expression, **(a=0, b=0, c=0)** will make the expression as given evaluate *false*. If the **&&** should have been **||**, the code has done the wrong thing and might branch to the wrong path.

---

4 Sections 23.4.3 through 23.4.7 have been adapted from an article by Brian Marick posted on the Internet newsgroup comp.testing. This adaptation is included with the permission of the author. For further discussion of these topics, see [MAR94].

5 The code presented in this and the following sections uses C++ syntax. For further information, see any good book on C++.

Of course, the effectiveness of these techniques depends on how testers perceive a "plausible fault." If real faults in an OO system are perceived to be "implausible," then this approach is really no better than any random testing technique. However, if the analysis and design models can provide insight into what is likely to go wrong, then fault-based testing can find significant numbers of errors with relatively low expenditures of effort.

? What types of faults are encountered in operation calls and message connections?

Integration testing looks for plausible faults in operation calls or message connections. Three types of faults are encountered in this context: unexpected result, wrong operation/message used, incorrect invocation. To determine plausible faults as functions (operations) are invoked, the behavior of the operation must be examined.

Integration testing applies to attributes as well as to operations. The "behaviors" of an object are defined by the values that its attributes are assigned. Testing should exercise the attributes to determine whether proper values occur for distinct types of object behavior.

It is important to note that integration testing attempts to find errors in the client object, not the server. Stated in conventional terms, the focus of integration testing is to determine whether errors exist in the calling code, not the called code. The operation call is used as a clue, a way to find test requirements that exercise the calling code.

23.4.4 The Impact of OO Programming on Testing

Quote:

"If you want and expect a program to work, you will be more likely to see a working program—you will miss failures."

Cem Kaner et al.

There are several ways object-oriented programming can have an impact on testing. Depending on the approach to OOP,

- Some types of faults become less plausible (not worth testing for).
- Some types of faults become more plausible (worth testing now).
- Some new types of faults appear.

When an operation is invoked, it may be hard to tell exactly what code gets exercised. That is, the operation may belong to one of many classes. Also, it can be hard to determine the exact type or class of a parameter. When the code accesses it, it may get an unexpected value. The difference can be understood by considering a conventional function call:

```
x = func (y);
```

For conventional software, the tester need consider all behaviors attributed to **func** and nothing more. In an OO context, the tester must consider the behaviors of **base::func()**, of **derived::func()**, and so on. Each time **func** is invoked, the tester must consider the union of all distinct behaviors. This is easier if good OO design practices are followed and the difference between superclasses and subclasses (in C++ jargon, these are called *base classes* and *derived classes*) are limited. The testing approach for base and derived classes is essentially the same. The difference is one of bookkeeping.

Testing OO class operations is analogous to testing code that takes a function parameter and then invokes it. Inheritance is a convenient way of producing polymorphic operations. At the call site, what matters is not the inheritance, but the polymorphism. Inheritance does make the search for test requirements more straightforward.

By virtue of OO software architecture and construction, are some types of faults more plausible for an OO system and others less plausible? The answer is, "Yes." For example, because OO operations are generally smaller, more time tends to be spent on integration because there are more opportunities for integration faults. Therefore, integration faults become more plausible.

23.4.5 Test Cases and the Class Hierarchy

As noted earlier in this chapter, inheritance does not obviate the need for thorough testing of all derived classes. In fact, it can actually complicate the testing process.

Consider the following situation. A class **base** contains operations *inherited* and *redefined*. A class **derived** redefines *redefined* to serve in a local context. There is little doubt the **derived::redefined()** has to be tested because it represents a new design and new code. But does **derived::inherited()** have to be retested?

Even though a base class has been thoroughly tested, you will still have to test all classes derived from it.

If **derived::inherited()** calls *redefined* and the behavior of *redefined* has changed, **derived::inherited()** may mishandle the new behavior. Therefore, it needs new tests even though the design and code have not changed. It is important to note, however, that only a subset of all tests for **derived::inherited()** may have to be conducted. If part of the design and code for *inherited* does not depend on *redefined* (i.e., that does not call it nor call any code that indirectly calls it), that code need not be retested in the derived class.

Base::redefined() and **derived::redefined()** are two different operations with different specifications and implementations. Each would have a set of test requirements derived from the specification and implementation. Those test requirements probe for plausible faults: integration faults, condition faults, boundary faults, and so forth. But the operations are likely to be similar. Their sets of test requirements will overlap. The better the OO design, the greater is the overlap. New tests need to be derived only for those **derived::redefined()** requirements that are not satisfied by the **base::redefined()** tests.

To summarize, the **base::redefined()** tests are applied to objects of class **derived**. Test inputs may be appropriate for both base and derived classes, but the expected results may differ in the derived class.

23.4.6 Scenario-Based Test Design

Fault-based testing misses two main types of errors: (1) incorrect specifications and (2) interactions among subsystems. When errors associated with incorrect specification occur, the product doesn't do what the customer wants. It might do the wrong

thing or it might omit important functionality. But in either circumstance, quality (conformance to requirements) suffers. Errors associated with subsystem interaction occur when the behavior of one subsystem creates circumstances (e.g., events, data flow) that cause another subsystem to fail.

Scenario-based testing will uncover errors that occur when any actor interacts with the OO software.

Scenario-based testing concentrates on what the user does, not what the product does. This means capturing the tasks (via use-cases) that the user has to perform, then applying them and their variants as tests.

Scenarios uncover interaction errors. But to accomplish this, test cases must be more complex and more realistic than fault-based tests. Scenario-based testing tends to exercise multiple subsystems in a single test (users do not limit themselves to the use of one subsystem at a time).

As an example, consider the design of scenario-based tests for a text editor. Use cases follow:

Use-Case: *Fix the Final Draft*

Background: It's not unusual to print the "final" draft, read it, and discover some annoying errors that weren't obvious from the on-screen image. This use-case describes the sequence of events that occurs when this happens.

1. Print the entire document.
2. Move around in the document, changing certain pages.
3. As each page is changed, it's printed.
4. Sometimes a series of pages is printed.

This scenario describes two things: a test and specific user needs. The user needs are obvious: (1) a method for printing single pages and (2) a method for printing a range of pages. As far as testing goes, there is a need to test editing after printing (as well as the reverse). The tester hopes to discover that the *printing* function causes errors in the *editing* function; that is, that the two software functions are not properly independent.

Although scenario-based testing has merit, you will get a higher return on time invested by reviewing use-cases when they are developed during OOA.

Use-Case: *Print a New Copy*

Background: Someone asks the user for a fresh copy of the document. It must be printed.

1. Open the document.
2. Print it.
3. Close the document.

Again, the testing approach is relatively obvious. Except that this document didn't appear out of nowhere. It was created in an earlier task. Does that task affect this one?

In many modern editors, documents remember how they were last printed. By default, they print the same way next time. After the *Fix the Final Draft* scenario, just selecting "Print" in the menu and clicking the "Print" button in the dialog box will cause the last corrected page to print again. So, according to the editor, the correct scenario should look like this:

Use-Case: *Print a New Copy*

1. Open the document.
2. Select "Print" in the menu.
3. Check if you're printing a page range; if so, click to print the entire document.
4. Click on the Print button.
5. Close the document.

But this scenario indicates a potential specification error. The editor does not do what the user reasonably expects it to do. Customers will often overlook the check noted in step 3 above. They will then be annoyed when they trot off to the printer and find one page when they wanted 100. Annoyed customers signal specification bugs.

A test case designer might miss this dependency in test design, but it is likely that the problem would surface during testing. The tester would then have to contend with the probable response, "That's the way it's supposed to work!"

23.4.7 Testing Surface Structure and Deep Structure

Surface structure refers to the externally observable structure of an OO program. That is, the structure that is immediately obvious to an end-user. Rather than performing functions, the users of many OO systems may be given objects to manipulate in some way. But whatever the interface, tests are still based on user tasks. Capturing these tasks involves understanding, watching, and talking with representative users (and as many nonrepresentative users as are worth considering).

Structure testing occurs at two levels: (1) tests that exercise the structure observable by the end-user and (2) tests designed to exercise the internal program structure.

There will surely be some difference in detail. For example, in a conventional system with a command-oriented interface, the user might use the list of all commands as a testing checklist. If no test scenarios existed to exercise a command, testing has likely overlooked some user tasks (or the interface has useless commands). In a object-based interface, the tester might use the list of all objects as a testing checklist.

The best tests are derived when the designer looks at the system in a new or unconventional way. For example, if the system or product has a command-based interface, more thorough tests will be derived if the test case designer pretends that operations are independent of objects. Ask questions like, "Might the user want to use this operation—which applies only to the **Scanner** object—while working with the printer?" Whatever the interface style, test case design that exercises the surface structure should use both objects and operations as clues leading to overlooked tasks.

Deep structure refers to the internal technical details of an OO program. That is, the structure that is understood by examining the design and/or code. Deep structure testing is designed to exercise dependencies, behaviors, and communication mechanisms that have been established as part of the system and object design (Chapter 22) of OO software.

The analysis and design models are used as the basis for deep structure testing. For example, the object-relationship diagram or the subsystem collaboration diagram depicts collaborations between objects and subsystems that may not be externally

visible. The test case design then asks: "Have we captured (as a test) some task that exercises the collaboration noted on the object-relationship diagram or the subsystem collaboration diagram? If not, why not?"

Design representations of class hierarchy provide insight into inheritance structure. Inheritance structure is used in fault-based testing. Consider a situation in which an operation named *caller* has only one argument and that argument is a reference to a base class. What might happen when *caller* is passed a derived class? What are the differences in behavior that could affect *caller?* The answers to these questions might lead to the design of specialized tests.

23.5 TESTING METHODS APPLICABLE AT THE CLASS LEVEL

In Chapter 17, we noted that software testing begins "in the small" and slowly progresses toward testing "in the large." Testing in the small focuses on a single class and the methods that are encapsulated by the class. Random testing and partitioning are methods that can be used to exercise a class during OO testing [KIR94].

23.5.1 Random Testing for OO Classes

To provide brief illustrations of these methods, consider a banking application in which an **account** class has the following operations: *open, setup, deposit, withdraw, balance, summarize, creditLimit,* and *close* [KIR94]. Each of these operations may be applied for **account**, but certain constraints (e.g., the account must be opened before other operations can be applied and closed after all operations are completed) are implied by the nature of the problem. Even with these constraints, there are many permutations of the operations. The minimum behavioral life history of an instance of **account** includes the following operations:

The number of possible permutations for random testing can grow quite large. A strategy similar to orthogonal array testing (Chapter 17) can be used to improve testing efficiency.

open • setup • deposit • withdraw • close

This represents the minimum test sequence for account. However, a wide variety of other behaviors may occur within this sequence:

open • setup • deposit • [deposit | withdraw | balance | summarize | creditLimit]^n • withdraw • close

A variety of different operation sequences can be generated randomly. For example:

Test case r_1: **open • setup • deposit • deposit • balance • summarize • withdraw • close**
Test case r_2: **open • setup • deposit • withdraw • deposit • balance • creditLimit • withdraw • close**

These and other random order tests are conducted to exercise different class instance life histories.

23.5.2 Partition Testing at the Class Level

Partition testing reduces the number of test cases required to exercise the class in much the same manner as equivalence partitioning (Chapter 17) for conventional

software. Input and output are categorized and test cases are designed to exercise each category. But how are the partitioning categories derived?

What testing options are available at the class level?

State-based partitioning categorizes class operations based on their ability to change the state of the class. Again considering the **account** class, state operations include *deposit* and *withdraw*, whereas nonstate operations include *balance, summarize,* and *creditLimit.* Tests are designed in a way that exercises operations that change state and those that do not change state separately. Therefore,

Test case p_1: **open • setup • deposit • deposit • withdraw • withdraw • close**
Test case p_2: **open • setup • deposit • summarize • creditLimit • withdraw • close**

Test case p_1 changes state, while test case p_2 exercises operations that do not change state (other than those in the minimum test sequence).

Attribute-based partitioning categorizes class operations based on the attributes that they use. For the **account** class, the attributes **balance** and **creditLimit** can be used to define partitions. Operations are divided into three partitions: (1) operations that use **creditLimit**, (2) operations that modify **creditLimit**, and (3) operations that do not use or modify **creditLimit**. Test sequences are then designed for each partition.

Category-based partitioning categorizes class operations based on the generic function that each performs. For example, operations in the **account** class can be categorized in initialization operations (*open, setup*), computational operations (*deposit, withdraw*). queries (*balance, summarize, creditLimit*) and termination operations (*close*).

23.6 INTERCLASS TEST CASE DESIGN

Test case design becomes more complicated as integration of the OO system begins. It is at this stage that testing of collaborations between classes must begin. To illustrate "interclass test case generation" [KIR94], we expand the banking example introduced in Section 23.5 to include the classes and collaborations noted in Figure 23.2. The direction of the arrows in the figure indicates the direction of messages and the labeling indicates the operations that are invoked as a consequence of the collaborations implied by the messages.

Like the testing of individual classes, class collaboration testing can be accomplished by applying random and partitioning methods, as well as scenario-based testing and behavioral testing.

23.6.1 Multiple Class Testing

Kirani and Tsai [KIR94] suggest the following sequence of steps to generate multiple class random test cases:

1. For each client class, use the list of class operations to generate a series of random test sequences. The operations will send messages to other server classes.

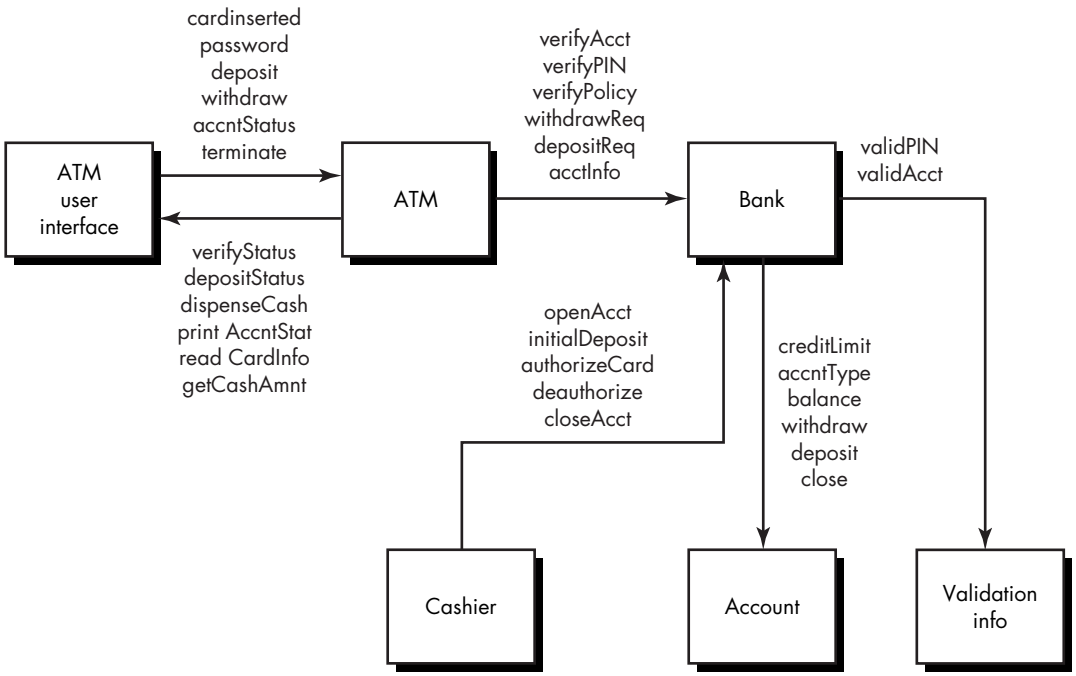


FIGURE 23.2 Class collaboration diagram for banking application [KIR94]

2. For each message that is generated, determine the collaborator class and the corresponding operation in the server object.
3. For each operation in the server object (that has been invoked by messages sent from the client object), determine the messages that it transmits.
4. For each of the messages, determine the next level of operations that are invoked and incorporate these into the test sequence.

To illustrate [KIR94], consider a sequence of operations for the **bank** class relative to an **ATM** class (Figure 23.2):

$$\text{verifyAcct} \bullet \text{verifyPIN} \bullet [[\text{verifyPolicy} \bullet \text{withdrawReq}] \mid \text{depositReq} \mid \text{acctInfoREQ}]^{n}$$

A random test case for the **bank** class might be

$$\text{test case } r_3 = \text{verifyAcct} \bullet \text{verifyPIN} \bullet \text{depositReq}$$

In order to consider the collaborators involved in this test, the messages associated with each of the operations noted in test case r_3 is considered. **Bank** must collaborate with **ValidationInfo** to execute the *verifyAcct* and *verifyPIN*. **Bank** must collaborate with **account** to execute *depositReq*. Hence, a new test case that exercises these collaborations is

$$\text{test case } r_4 = \text{verifyAcct}_{\text{Bank}}[\text{validAcct}_{\text{ValidationInfo}}] \bullet \text{verifyPIN}_{\text{Bank}} \bullet [\text{validPin}_{\text{ValidationInfo}}] \bullet \text{depositReq} \bullet [\text{deposit}_{\text{account}}]$$

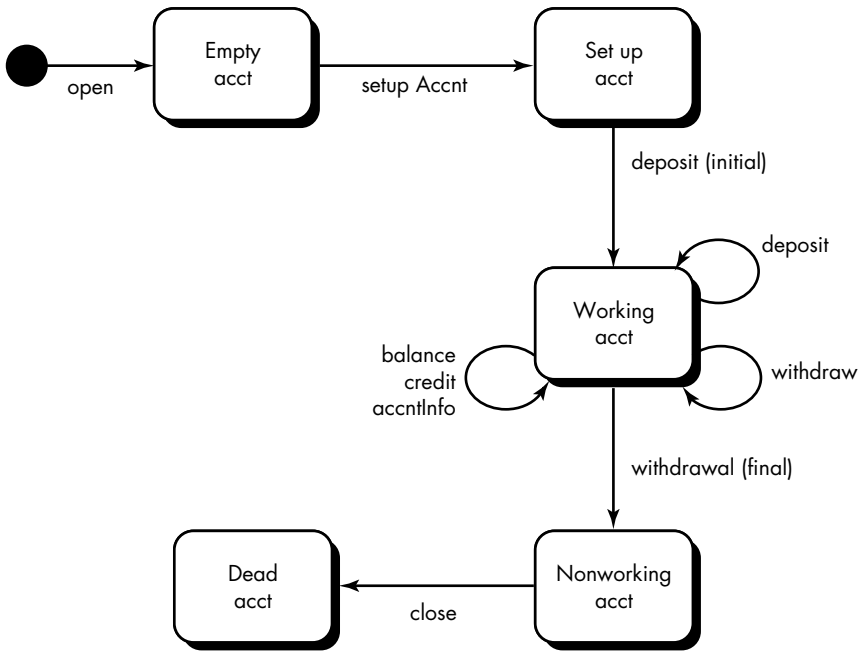


FIGURE 23.3
State transition diagram for account class [KIR94]

The approach for multiple class partition testing is similar to the approach used for partition testing of individual classes. A single class is partitioned as discussed in Section 23.4.5. However, the test sequence is expanded to include those operations that are invoked via messages to collaborating classes. An alternative approach partitions tests based on the interfaces to a particular class. Referring to Figure 23.2, the **bank** class receives messages from the **ATM** and **cashier** classes. The methods within **bank** can therefore be tested by partitioning them into those that serve **ATM** and those that serve **cashier**. State-based partitioning (Section 23.4.9) can be used to refine the partitions further.

23.6.2 Tests Derived from Behavior Models

In Chapter 21, we discussed the use of the state transition diagram as a model that represents the dynamic behavior of a class. The STD for a class can be used to help derive a sequence of tests that will exercise the dynamic behavior of the class (and those classes that collaborate with it). Figure 23.3 [KIR94] illustrates an STD for the **account** class discussed earlier.[6] Referring to the figure, initial transitions move through the *empty acct* and *setup acct* states. The majority of all behavior for instances of the class occurs while in the *working acct* state. A final withdrawal and close cause the account class to make transitions to the *nonworking acct* and *dead acct* states, respectively.

6 UML symbology is used for the STD shown in Figure 23.3. It differs slightly from the symbology used for STDs in Part Three of this book.

The tests to be designed should achieve all state coverage [KIR94]. That is, the operation sequences should cause the **account** class to make transition through all allowable states:

test case s_1: open • setupAccnt • deposit (initial) • withdraw (final) • close

It should be noted that this sequence is identical to the minimum test sequence discussed in Section 23.5.1. Adding additional test sequences to the minimum sequence,

test case s_2: open • setupAccnt • deposit(initial) • deposit • balance • credit • withdraw (final) • close
test case s_3: open • setupAccnt • deposit(initial) • deposit • withdraw • accntInfo • withdraw (final) • close

Still more test cases could be derived to ensure that all behaviors for the class have been adequately exercised. In situations in which the class behavior results in a collaboration with one or more classes, multiple STDs are used to track the behavioral flow of the system.

The state model can be traversed in a "breadth-first" [MGR94] manner. In this context, *breadth first* implies that a test case exercise a single transition and that when a new transition is to be tested only previously tested transitions are used.

Consider the **credit card** object discussed in Section 23.2.2. The initial state of **credit card** is *undefined* (i.e., no credit card number has been provided). Upon reading the credit card during a sale, the object takes on a *defined* state; that is, the attributes **card number** and **expiration date**, along with bank specific identifiers are defined. The credit card is *submitted* when it is sent for authorization and it is *approved* when authorization is received. The transition of **credit card** from one state to another can be tested by deriving test cases that cause the transition to occur. A breadth-first approach to this type of testing would not exercise *submitted* before it exercised *undefined* and *defined*. If it did, it would make use of transitions that had not been previously tested and would therefore violate the breadth-first criterion.

23.7 SUMMARY

The overall objective of object-oriented testing—to find the maximum number of errors with a minimum amount of effort—is identical to the objective of conventional software testing. But the strategy and tactics for OO testing differ significantly. The view of testing broadens to include the review of both the analysis and design model. In addition, the focus of testing moves away from the procedural component (the module) and toward the class.

Because the OO analysis and design models and the resulting source code are semantically coupled, testing (in the form of formal technical reviews) begins during these engineering activities. For this reason, the review of CRC, object-relationship, and object-behavior models can be viewed as first stage testing.

Once OOP has been accomplished, unit testing is applied for each class. The design of tests for a class uses a variety of methods: fault-based testing, random testing, and partition testing. Each of these methods exercises the operations encapsulated by the class. Test sequences are designed to ensure that relevant operations are exercised. The state of the class, represented by the values of its attributes, is examined to determine if errors exist.

Integration testing can be accomplished using a thread-based or use-based strategy. Thread-based testing integrates the set of classes that collaborate to respond to one input or event. Use-based testing constructs the system in layers, beginning with those classes that do not use server classes. Integration test case design methods can also use random and partition tests. In addition, scenario-based testing and tests derived from behavioral models can be used to test a class and its collaborators. A test sequence tracks the flow of operations across class collaborations.

OO system validation testing is black-box oriented and can be accomplished by applying the same black-box methods discussed for conventional software. However, scenario-based testing dominates the validation of OO systems, making the use-case a primary driver for validation testing.

REFERENCES

[AMB95] Ambler, S., "Using Use Cases," *Software Development,* July 1995, pp. 53–61.

[BER93] Berard, E.V., *Essays on Object-Oriented Software Engineering,* vol. 1, Addison-Wesley, 1993.

[BIN94a] Binder, R.V., "Testing Object-Oriented Systems: A Status Report," *American Programmer,* vol. 7, no. 4, April 1994, pp. 23–28.

[BIN94b] Binder, R.V., "Object-Oriented Software Testing," *CACM,* vol. 37, no. 9, September 1994, p. 29.

[KIR94] Kirani, S. and W.T. Tsai, "Specification and Verification of Object-Oriented Programs," Technical Report TR 94-64, Computer Science Department, University of Minnesota, December 1994.

[LIN94] Lindland, O.I., et al., "Understanding Quality in Conceptual Modeling," *IEEE Software,* vol. 11, no 4, July 1994, pp. 42–49.

[MAR94] Marick, B., *The Craft of Software Testing,* Prentice-Hall, 1994.

[MGR94] McGregor, J.D. and T.D. Korson, "Integrated Object-Oriented Testing and Development Processes," *CACM,* vol. 37, no. 9, September 1994, pp. 59–77.

PROBLEMS AND POINTS TO PONDER

23.1. In your own words, describe why the class is the smallest reasonable unit for testing within an OO system.

23.2. Why do we have to retest subclasses that are instantiated from an existing class, if the existing class has already been thoroughly tested? Can we use the test cases designed for the existing class?

23.3. Why should "testing" begin with the OOA and OOD activities?

23.4. Derive a set of CRC index cards for *SafeHome* and conduct the steps noted in Section 23.2.2 to determine if inconsistencies exist.

23.5. What is the difference between thread-based and use-based strategies for integration testing? How does cluster testing fit in?

23.6. Apply random testing and partitioning to three classes defined in the design for the *SafeHome* system that you produced for Problem 22.12. Produce test cases that indicate the operation sequences that will be invoked.

23.7. Apply multiple class testing and tests derived from the behavioral model to the *SafeHome* design.

23.8. Derive tests using methods noted in Problems 23.6 and 23.7 for the PHTRS system described in Problem 12.13.

23.9. Derive tests using methods noted in Problems 23.6 and 23.7 for the video game considered in Problem 22.14.

23.10. Derive tests using methods noted in Problems 23.6 and 23.7 for the e-mail system considered in Problem 22.15.

23.11. Derive tests using methods noted in Problems 23.6 and 23.7 for the ATC system considered in Problem 22.16.

23.12. Derive four additional tests using each of the methods noted in Problems 23.6 and 23.7 for the banking application presented in Sections 23.5 and 23.6.

FURTHER READINGS AND INFORMATION SOURCES

The literature for object-oriented testing is relatively sparse, although it has expanded somewhat in recent years. Binder (*Testing Object-Oriented Systems: Models, Patterns, and Tools,* Addison-Wesley, 2000) has written the most comprehensive treatment of the subject published to date. Siegel and Muller (*Object Oriented Software Testing: A Hierarchical Approach,* Wiley, 1996) proposed a practical testing strategy for OO systems. Marick (*The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing,* Prentice-Hall, 1995) covers testing for both conventional and OO software.

Anthologies of important papers on OO testing have been edited by Kung et al. (*Testing Object-Oriented Software,* IEEE Computer Society, 1998) and Braude (*Object Oriented Analysis, Design and Testing: Selected Readings,* IEEE Computer Society, 1998).

These IEEE tutorials provide an interesting historical perspective on development in OO testing.

Jorgensen *(Software Testing: A Craftsman's Approach,* CRC Press, 1995) and McGregor and Sykes (*Object-Oriented Software Development,* Van Nostrand-Reinhold, 1992) present chapters dedicated to the topic. Beizer (*Black-Box Testing,* Wiley, 1995) discusses a variety of test case design methods that are appropriate in an OO context. Binder (*Testing Object-Oriented Systems,* Addison-Wesley, 1996) and Marick [MAR94] present detailed treatments of OO testing. In addition, many of the sources noted for Chapter 17 are generally applicable to OO testing.

A wide variety of information sources on object-oriented testing and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to OO testing can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/OOT.mhtml

CHAPTER

24 TECHNICAL METRICS FOR OBJECT-ORIENTED SYSTEMS

KEY CONCEPTS

Early in this book we noted that measurement and metrics are key components of any engineering discipline—and object-oriented software engineering is no exception. Sadly, the use of metrics for OO systems has progressed much more slowly than the use of other OO methods. Ed Berard [BER95] notes the irony of measurement when he states:

> Software people seem to have a love-hate relationship with metrics. On one hand, they despise and distrust anything that sounds or looks like a measurement. They are quick to point out the "flaws" in the arguments of anyone who talks about measuring software products, software processes, and (especially) software people. On the other hand, these same people seem to have no problems identifying which programming language is the best, the stupid things that managers do to "ruin" projects, and who's methodology works in what situations.

The "love-hate relationship" that Berard notes is real. And yet, as OO systems become more pervasive, it is essential that software engineers have quantitative measurements for assessing the quality of designs at both the architectural and component levels. These measures enable an engineer to assess the software early in the process, making changes that will reduce complexity and improve the long-term viability of the end product.

QUICK LOOK

What is it? Building OO software has been an engineering activity that relies more on collective wisdom, folklore, and qualitative guidance than on quantitative evaluation. OO metrics have been introduced to help a software engineer use quantitative analysis to assess the quality of the design before a system is built. The focus of OO metrics is on the class—the fundamental building block of the OO architecture.

Who does it? Software engineers use OO metrics to help them build higher-quality software.

Why is it important? As we stated in the Quick Look for Chapter 19, qualitative assessment of computer software must be complemented with quantitative analysis. A software engineer needs objective criteria to help guide the design of the OO architecture, the classes and subsystems that populate the architecture, and the operations and attributes that constitute a class. The tester needs quantitative guidance that will help in the selection of test cases and their targets. Technical metrics provide a basis from which analysis, design, and testing can be conducted more objectively and assessed more quantitatively.

What are the steps? The first step in the measurement process is to derive the software measures and metrics that are appropriate for the representation of software that is being considered. Next, data required to derive the formulated metrics are ▶

QUICK LOOK

collected. Once computed, appropriate metrics are analyzed based on pre-established guidelines and past data. The results of the analysis are interpreted to gain insight into the quality of the software, and the results of the interpretation lead to modification of work products arising out of analysis, design, code, or test.

What is the work product? Software metrics that are computed using data collected from the analysis and design models, source code, and test cases.

How do I ensure that I've done it right? You should establish the objectives of measurement before the data collection begins, defining each OO metric in an unambiguous manner. Define only a few metrics and then use them to gain insight into the quality of a software engineering work product.

24.1 THE INTENT OF OBJECT-ORIENTED METRICS

The primary objectives for object-oriented metrics are no different than those for metrics derived for conventional software:

- to better understand the quality of the product
- to assess the effectiveness of the process
- to improve the quality of work performed at a project level

Each of these objectives is important, but for the software engineer, product quality must be paramount. But how do we measure the quality of an OO system? What characteristics of the design model can be assessed to determine whether the system will be easy to implement, amenable to test, simple to modify, and most important, acceptable to end-users? These questions are addressed throughout the remainder of this chapter.

24.2 THE DISTINGUISHING CHARACTERISTICS OF OBJECT-ORIENTED METRICS

Metrics for any engineered product are governed by the unique characteristics of the product. For example, it would be meaningless to compute miles per gallon for an electric automobile. The metric is sound for conventional (i.e., gasoline powered) cars but it does not apply when the mode of propulsion changes radically. Object-oriented software is fundamentally different than software developed using conventional methods. For this reason, the metrics for OO systems must be tuned to the characteristics that distinguish OO from conventional software.

Berard [BER95] defines five characteristics that lead to specialized metrics: localization, encapsulation, information hiding, inheritance, and object abstraction techniques. Each of these characteristics is discussed briefly in the sections that follow.[1]

1 This discussion has been adapted from [BER95].

24.2.1 Localization

Localization is a characteristic of software that indicates the manner in which information is concentrated within a program. For example, conventional methods for functional decomposition localize information around functions, which are typically implemented as procedural modules. Data-driven methods localize information around specific data structures. In the OO context, information is concentrated by encapsulating both data and process within the bounds of a class or object.

XRef

Technical metrics for conventional software are discussed in Chapter 19.

Because conventional software emphasizes function as a localization mechanism, software metrics have focused on the internal structure or complexity of functions (e.g., module length, cohesion or cyclomatic complexity) or the manner in which functions connect to one another (e.g., module coupling).

Since the class is the basic unit of an OO system, localization is based on objects. Therefore, metrics should apply to the class (object) as a complete entity. In addition, the relationship between operations (functions) and classes is not necessarily one to one. Therefore, metrics that reflect the manner in which classes collaborate must be capable of accommodating one-to-many and many-to-one relationships.

24.2.2 Encapsulation

Berard [BER95] defines encapsulation as "the packaging (or binding together) of a collection of items. Low-level examples of encapsulation [for conventional software] include records and arrays, [and] subprograms (e.g., procedures, functions, subroutines, and paragraphs) are mid-level mechanisms for encapsulation."

XRef

Basic design concepts are discussed in Chapter 13. Their application to OO software is discussed in Chapter 20.

For OO systems, encapsulation encompasses the responsibilities of a class, including its attributes (and other classes for aggregate objects) and operations, and the states of the class, as defined by specific attribute values.

Encapsulation influences metrics by changing the focus of measurement from a single module to a package of data (attributes) and processing modules (operations). In addition encapsulation encourages measurement at a higher level of abstraction. For example, later in this chapter metrics associated with the number of operations per class will be introduced. Contrast this level of abstraction to conventional metrics that focus on counts of Boolean conditions (cyclomatic complexity) or line of code counts.

24.2.3 Information Hiding

Information hiding suppresses (or hides) the operational details of a program component. Only the information necessary to access the component is provided to those other components that wish to access it.

A well-designed OO system should encourage information hiding. Therefore, metrics that provide an indication of the degree to which hiding has been achieved should provide an indication of the quality of the OO design.

24.2.4 Inheritance

Inheritance is a mechanism that enables the responsibilities of one object to be propagated to other objects. Inheritance occurs throughout all levels of a class hierarchy. In general, conventional software does not support this characteristic.

Because inheritance is a pivotal characteristic in many OO systems, many OO metrics focus on it. Examples (discussed later in this chapter) include number of *children* (number of immediate instances of a class), number of *parents* (number of immediate generalizations), and class hierarchy *nesting level* (depth of a class in an inheritance hierarchy).

24.2.5 Abstraction

Abstraction is a mechanism that enables the designer to focus on the essential details of a program component (either data or process) with little concern for lower-level details. As Berard states: "Abstraction is a relative concept. As we move to higher levels of abstraction we ignore more and more details, i.e., we provide a more general view of a concept or item. As we move to lower levels of abstraction, we introduce more details, i.e., we provide a more specific view of a concept or item."

Because a class is an abstraction that can be viewed at many different levels of detail and in a number of different ways (e.g., as a list of operations, as a sequence of states, as a series of collaborations), OO metrics represent abstractions in terms of measures of a class (e.g., number of instances per class per application, number or parameterized classes per application, and ratio of parameterized classes to nonparameterized classes).

24.3 METRICS FOR THE OO DESIGN MODEL

There is much about object-oriented design that is subjective—an experienced designer "knows" how to characterize an OO system so that it will effectively implement customer requirements. But, as an OO design model grows in size and complexity, a more objective view of the characteristics of the design can benefit both the experienced designer (who gains additional insight) and the novice (who obtains an indication of quality that would otherwise be unavailable).

In a detailed treatment of software metrics for OO systems, Whitmire [WHI97] describes nine distinct and measurable characteristics of an OO design:

? What characteristics can be measured when we assess an OO design?

Size. Size is defined in terms of four views: population, volume, length, and functionality. *Population* is measured by taking a static count of OO entities such as classes or operations. *Volume* measures are identical to population measures but are collected dynamically—at a given instant of time. *Length* is a measure of a chain of interconnected design elements (e.g., the depth of an inheritance tree is a measure of length). *Functionality* metrics provide an indirect indication of the value delivered to the customer by an OO application.

Complexity. Like size, there are many differing views of software complexity [ZUS97]. Whitmire views complexity in terms of structural characteristics by examining how classes of an OO design are interrelated to one another.

Coupling. The physical connections between elements of the OO design (e.g., the number of collaborations between classes or the number of messages passed between objects) represent coupling within an OO system.

Sufficiency. Whitmire defines *sufficiency* as "the degree to which an abstraction possesses the features required of it, or the degree to which a design component possesses features in its abstraction, from the point of view of the current application." Stated another way, we ask: "What properties does this abstraction (class) need to possess to be useful to me?" [WHI97]. In essence, a design component (e.g., a class) is sufficient if it fully reflects all properties of the application domain object that it is modeling—that is, that the abstraction (class) possesses the features required of it.

Quote:

"Many of the design decisions for which I had to rely on folklore and myth can now be made using quantitative data."

Scott Whitmire

Completeness. The only difference between completeness and sufficiency is "the feature set against which we compare the abstraction or design component [WHI97]." Sufficiency compares the abstraction from the point of view of the current application. Completeness considers multiple points of view, asking the question: "What properties are required to fully represent the problem domain object?" Because the criterion for completeness considers different points of view, it has an indirect implication about the degree to which the abstraction or design component can be reused.

Cohesion. Like its counterpart in conventional software, an OO component should be designed in a manner that has all operations working together to achieve a single, well-defined purpose. The cohesiveness of a class is determined by examining the degree to which "the set of properties it possesses is part of the problem or design domain" [WHI97].

WebRef

A NASA technical report addressing quality metrics for OO systems can be downloaded from **satc.gsfc.nasa.gov/support/index.html**

Primitiveness. A characteristic that is similar to simplicity, primitiveness (applied to both operations and classes) is the degree to which an operation is *atomic*—that is, the operation cannot be constructed out of a sequence of other operations contained within a class. A class that exhibits a high degree of primitiveness encapsulates only primitive operations.

Similarity. The degree to which two or more classes are similar in terms of their structure, function, behavior, or purpose is indicated by this measure.

Volatility. As we have seen earlier in this book, design changes can occur when requirements are modified or when modifications occur in other parts of an application, resulting in mandatory adaptation of the design component in question. Volatility of an OO design component measures the likelihood that a change will occur.

Whitmire's derivation of metrics for these design characteristics is beyond the scope of this book. Interested readers should see [WHI97] for more detail.

In reality, technical metrics for OO systems can be applied not only to the design model, but also the analysis model. In the sections that follow, we explore metrics that provide an indication of quality at the OO class level and the operation level. In addition, metrics applicable for project management and testing are also explored.

24.4 CLASS-ORIENTED METRICS

The class is the fundamental unit of an OO system. Therefore, measures and metrics for an individual class, the class hierarchy, and class collaborations will be invaluable to a software engineer who must assess design quality. In earlier chapters, we saw that the class encapsulates operations (processing) and attributes (data). The class is often the "parent" for subclasses (sometimes called *children*) that inherit its attributes and operations. The class often collaborates with other classes. Each of these characteristics can be used as the basis for measurement.[2]

24.4.1 The CK Metrics Suite

One of the most widely referenced sets of OO software metrics has been proposed by Chidamber and Kemerer [CHI94]. Often referred to as the *CK metrics suite,* the authors have proposed six class-based design metrics for OO systems.[3]

Weighted methods per class (WMC). Assume that n methods of complexity c_1, c_2, . . ., c_n are defined for a class **C**. The specific complexity metric that is chosen (e.g., cyclomatic complexity) should be normalized so that nominal complexity for a method takes on a value of 1.0.

$$\text{WMC} = \Sigma\, c_i$$

for $i = 1$ to n. The number of methods and their complexity are reasonable indicators of the amount of effort required to implement and test a class. In addition, the larger the number of methods, the more complex is the inheritance tree (all subclasses inherit the methods of their parents). Finally, as the number of methods grows for a given class, it is likely to become more and more application specific, thereby limiting potential reuse. For all of these reasons, WMC should be kept as low as is reasonable.

KEY POINT

The number of methods and their complexity are directly correlated to the effort required to test a class.

2 It should be noted that the validity of some of the metrics discussed in this chapter is currently debated in the technical literature. Those who champion measurement theory demand a degree of formalism that some of the OO metrics do not provide. However, it is reasonable to state that all of the metrics noted provide useful insight for the software engineer.

3 Chidamber, Darcy, and Kemerer use the term *methods* rather than *operations*. Their usage of the term is reflected in this section.

Although it would seem relatively straightforward to develop a count for the number of methods in a class, the problem is actually more complex than it seems. Churcher and Shepperd [CHU95] discuss this issue when they write:

> In order to count methods, we must answer the fundamental question "Does a method belong only to the class which defines it, or does it also belong to every class which inherits it directly or indirectly?" Questions such as this may seem trivial since the runtime system will ultimately resolve them. However, the implications for metrics may be significant.
>
> One possibility is to restrict counting to the current class, ignoring inherited members. The motivation for this would be that inherited members have already been counted in the classes where they are defined, so the class increment is the best measure of its functionality—what it does reflects its reason for existing. In order to understand what a class does, the most important source of information is its own operations. If a class cannot respond to a message (i.e., it lacks a corresponding method of its own) then it will pass the message on to its parent(s).
>
> At the other extreme, counting could include all methods defined in the current class, together with all inherited methods. This approach emphasizes the importance of the state space, rather than the class increment, in understanding a class.
>
> Between these extremes lie a number of other possibilities. For example, one could restrict counting to the current class and members inherited directly from parent(s). This approach would be based on the argument that the specialization of parent classes is the most directly relevant to the behavior of a child class.

Like most counting conventions in software metrics, any of the approaches just outlined is acceptable, as long as the counting approach is applied consistently whenever metrics are collected.

Depth of the inheritance tree (DIT). This metric is "the maximum length from the node to the root of the tree" [CHI94]. Referring to Figure 24.1, the value of DIT for the class-hierarchy shown is 4. As DIT grows, it is likely that lower-level classes will inherit many methods. This leads to potential difficulties when attempting to predict the behavior of a class. A deep class hierarchy (DIT is large) also leads to greater design complexity. On the positive side, large DIT values imply that many methods may be reused.

Inheritance is an extremely powerful feature that can get you into trouble, if you use it without care. Use DIT and other related metrics to give yourself a reading on the complexity of class hierarchies.

Number of children (NOC). The subclasses that are immediately subordinate to a class in the class hierarchy are termed its *children*. Referring to Figure 24.1, class $\mathbf{C_2}$ has three children—subclasses $\mathbf{C_{21}}$, $\mathbf{C_{22}}$, and $\mathbf{C_{23}}$. As the number of children grows, reuse increases but also, as NOC increases, the abstraction represented by the parent class can be diluted. That is, some of the children may not really be appropriate members of the parent class. As NOC increases, the amount of testing (required to exercise each child in its operational context) will also increase.

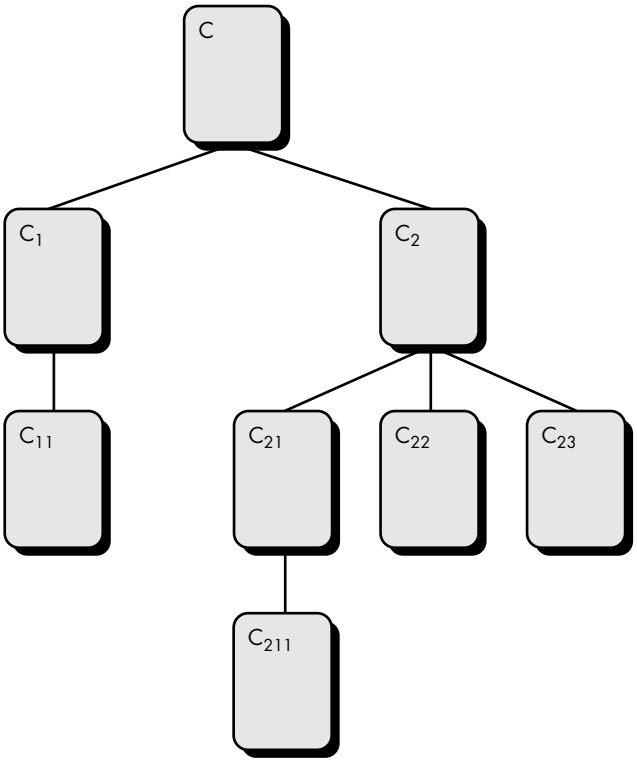


FIGURE 24.1 A class hierarchy

Coupling between object classes (CBO). The CRC model (Chapter 21) may be used to determine the value for CBO. In essence, CBO is the number of collaborations listed for a class on its CRC index card. As CBO increases, it is likely that the reusability of a class will decrease. High values of CBO also complicate modifications and the testing that ensues when modifications are made. In general, the CBO values for each class should be kept as low as is reasonable. This is consistent with the general guideline to reduce coupling in conventional software.

The concepts of coupling and cohesion apply to both conventional and OO software. Keep class coupling low and class and operation cohesion high.

Response for a class (RFC). The response set of a class is "a set of methods that can potentially be executed in response to a message received by an object of that class" [CHI94]. RFC is the number of methods in the response set. As RFC increases, the effort required for testing also increases because the test sequence (Chapter 23) grows. It also follows that, as RFC increases, the overall design complexity of the class increases.

Lack of cohesion in methods (LCOM). Each method within a class, **C,** accesses one or more attributes (also called *instance variables*). LCOM is the number of methods that access one or more of the same attributes.[4] If no methods access the same

4 The formal definition is a bit more complex. See [CHI94] for details.

attributes, then LCOM = 0. To illustrate the case where LCOM ≠ 0, consider a class with six methods. Four of the methods have one or more attributes in common (i.e., they access common attributes). Therefore, LCOM = 4. If LCOM is high, methods may be coupled to one another via attributes. This increases the complexity of the class design. In general, high values for LCOM imply that the class might be better designed by breaking it into two or more separate classes. Although there are cases in which a high value for LCOM is justifiable, it is desirable to keep cohesion high; that is, keep LCOM low.

Quote:

"Object-oriented measures are an integral part of object technology and of good software engineering."

Brian Henderson-Sellers

24.4.2 Metrics Proposed by Lorenz and Kidd

In their book on OO metrics, Lorenz and Kidd [LOR94] divide class-based metrics into four broad categories: size, inheritance, internals, and externals. Size-oriented metrics for the OO class focus on counts of attributes and operations for an individual class and average values for the OO system as a whole. Inheritance-based metrics focus on the manner in which operations are reused through the class hierarchy. Metrics for class internals look at cohesion (Section 24.4.1) and code-oriented issues, and external metrics examine coupling and reuse. A sampling of metrics proposed by Lorenz and Kidd follows:[5]

Class size (CS). The overall size of a class can be measured by determining the following measures:

- The total number of operations (both inherited and private instance operations) that are encapsulated within the class.
- The number of attributes (both inherited and private instance attributes) that are encapsulated by the class.

ADVICE

During review of the OOA model, the CRC index cards will provide a reasonable indication of expected values for CS. If you encounter a class with a large responsibility count during OOA, consider partitioning it.

The WMC metric proposed by Chidamber and Kemerer (Section 24.4.1) is also a weighted measure of class size. As we noted earlier, large values for CS indicate that a class may have too much responsibility. This will reduce the reusability of the class and complicate implementation and testing. In general, inherited or public operations and attributes should be weighted more heavily in determining class size [LOR94]. Private operations and attributes enable specialization and are more localized in the design. Averages for the number of class attributes and operations may also be computed. The lower the average values for size, the more likely that classes within the system can be reused widely.

Number of operations overridden by a subclass (NOO). There are instances when a subclass replaces an operation inherited from its superclass with a specialized version

5 For a complete discussion, see [LOR94].

for its own use. This is called *overriding*. Large values for NOO generally indicate a design problem. As Lorenz and Kidd point out:

> Since a subclass should be a specialization of its superclasses, it should primarily extend the services [operations] of the superclasses. This should result in unique new method names.

If NOO is large, the designer has violated the abstraction implied by the superclass. This results in a weak class hierarchy and OO software that can be difficult to test and modify.

Number of operations added by a subclass (NOA). Subclasses are specialized by adding private operations and attributes. As the value for NOA increases, the subclass drifts away from the abstraction implied by the superclass. In general, as the depth of the class hierarchy increases (DIT becomes large), the value for NOA at lower levels in the hierarchy should go down.

Specialization index (SI). The specialization index provides a rough indication of the degree of specialization for each of the subclasses in an OO system. Specialization can be achieved by adding or deleting operations or by overriding.

$$SI = [NOO \times level]/M_{total}$$

where *level* is the level in the class hierarchy at which the class resides and M_{total} is the total number of methods for the class. The higher is the value of SI, the more likely the class hierarchy has classes that do not conform to the superclass abstraction.

24.4.3 The MOOD Metrics Suite

Harrison, Counsell, and Nithi [HAR98] propose a set of metrics for object-oriented design that provide quantitative indicators for OO design characteristics. A sampling of MOOD metrics follows:

Quote:

"Analyzing OO software in order to evaluate its quality is becoming increasingly important as the paradigm continues to increase in popularity."

Rachel Harrison, et al.

Method inheritance factor (MIF). The degree to which the class architecture of an OO system makes use of inheritance for both methods (operations) and attributes is defined as

$$MIF = \Sigma\, M_i(C_i)/\Sigma\, M_a(C_i)$$

where the summation occurs over $i = 1$ to TC. TC is defined as the total number of classes in the architecture, C_i is a class within the architecture, and

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

where

$M_a(C_i)$ = the number of methods that can be invoked in association with C_i.
$M_d(C_i)$) = the number of methods declared in the class C_i.
$M_i(C_i)$ = the number of methods inherited (and not overridden) in C_i.

The value of MIF (the *attribute inheritance factor,* AIF, is defined in an analogous manner) provides an indication of the impact of inheritance on the OO software.

Coupling factor (CF). Earlier in this chapter we noted that coupling is an indication of the connections between elements of the OO design. The MOOD metrics suite defines coupling in the following way:

$$\text{CF} = \Sigma_i \Sigma_j \textit{is\_client}\ (C_i, C_j)]/(\text{TC}^2 - \text{TC})$$

where the summations occur over $i = 1$ to TC and $j = 1$ to TC. The function

is_client = 1, *if and only if* a relationship exists between the client class, C_c, and the server class, C_s, and $C_c \neq C_s$
= 0, otherwise

Although many factors affect software complexity, understandability, and maintainability, it is reasonable to conclude that, as the value for CF increases, the complexity of the OO software will also increase and understandability, maintainability, and the potential for reuse may suffer as a result.

Polymorphism factor (PF). Harrison and her colleagues [HAR98] define PF as "the number of methods that redefine inherited methods, divided by the maximum number of possible distinct polymorphic situations . . . [t]hus, PF is an indirect measure of the relative amount of dynamic binding in a system." The MOOD metrics suite defines PF in the following manner:

$$\text{MIF} = \Sigma_i M_o(C_i)/\Sigma_i [M_n(C_i) \times \text{DC}(C_i)]$$

where the summations occur over $i = 1$ to TC and

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

Also,

$M_n(C_i)$ = the number of new methods.
$M_o(C_i)$ = the number of overriding methods.
$\text{DC}(C_i)$ = the descendants count (the number of descendant classes of a base class).

Harrison and her colleagues [HAR98] present a detailed analysis of MIF, CF, and PF along with other metrics and examine their validity for use in the assessment of design quality.

24.5 OPERATION-ORIENTED METRICS

Because the class is the dominant unit in OO systems, fewer metrics have been proposed for operations that reside within a class. Churcher and Shepperd [CHU95] discuss this when they state:

> Results of recent studies indicate that methods tend to be small, both in terms of number of statements and in logical complexity [WIL93], suggesting that connectivity structure of a system may be more important than the content of individual modules.

However, some insights can be gained by examining average characteristics for methods (operations). Three simple metrics, proposed by Lorenz and Kidd [LOR94], are noted next:

XRef

Metrics that can be applied at the component level can also be applied to operations. See Chapter 19 for details.

Average operation size (OS_{avg}). Although lines of code could be used as an indicator for operation size, the LOC measure suffers from all the problems discussed in Chapter 4. For this reason, the number of messages sent by the operation provides an alternative for operation size. As the number of messages sent by a single operation increases, it is likely that responsibilities have not been well-allocated within a class.

Operation complexity (OC). The complexity of an operation can be computed using any of the complexity metrics (Chapter 19) proposed for conventional software [ZUS90]. Because operations should be limited to a specific responsibility, the designer should strive to keep OC as low as possible.

Average number of parameters per operation (NP_{avg}). The larger the number of operation parameters, the more complex the collaboration between objects. In general, NP_{avg} should be kept as low as possible.

24.6 METRICS FOR OBJECT-ORIENTED TESTING

The design metrics noted in Sections 24.4 and 24.5 provide an indication of design quality. They also provide a general indication of the amount of testing effort required to exercise an OO system.

Binder [BIN94] suggests a broad array of design metrics that have a direct influence on the "testability" of an OO system. The metrics are organized into categories that reflect important design characteristics.

Encapsulation

Lack of cohesion in methods (LCOM).[6] The higher the value of LCOM, the more states must be tested to ensure that methods do not generate side effects.

6 See Section 24.4.1 for a description of LCOM.

Percent public and protected (PAP). Public attributes are inherited from other classes and therefore visible to those classes. Protected attributes are a specialization and private to a specific subclass. This metric indicates the percentage of class attributes that are public. High values for PAP increase the likelihood of side effects among classes. Tests must be designed to ensure that such side effects are uncovered.

OO testing can be quite complex. Metrics can assist you in targeting testing resources at threads, scenarios, and class clusters that are "suspect" based on measured characteristics. Use them.

Public access to data members (PAD). This metric indicates the number of classes (or methods) that can access another class's attributes, a violation of encapsulation. High values for PAD lead to the potential for side effects among classes. Tests must be designed to ensure that such side effects are uncovered.

Inheritance

Number of root classes (NOR). This metric is a count of the distinct class hierarchies that are described in the design model. Test suites for each root class and the corresponding class hierarchy must be developed. As NOR increases, testing effort also increases.

Fan-in (FIN). When used in the OO context, fan-in is an indication of multiple inheritance. FIN > 1 indicates that a class inherits its attributes and operations from more than one root class. FIN > 1 should be avoided when possible.

Number of children (NOC) and depth of the inheritance tree (DIT).[7] As we discussed in Chapter 23, superclass methods will have to be retested for each subclass.

In addition to these metrics, Binder [BIN94] defines metrics for class complexity and polymorphism. The metrics defined for class complexity include three CK metrics (Section 24.4.1): weighted methods per class, coupling between object classes, and response for a class. In addition, metrics associated with method counts are defined. The metrics associated with polymorphism are highly specialized. A discussion of them is best left to Binder.

24.7 METRICS FOR OBJECT-ORIENTED PROJECTS

As we discovered in Part Two of this book, the job of the project manager is to plan, coordinate, track, and control a software project. In Chapter 20, we discussed some of the special issues associated with management of OO projects. But what about measurement? Are there specialized OO metrics that can be used by the project manager to provide additional insight into progress?[8] The answer, of course, is, "Yes."

---

7 See Section 24.4.1 for a description of NOC and DIT.

8 A worthwhile discussion of the CK metrics suite (Section 24.4.1) for use in management decision-making can be found in [CHI98].

XRef

The applicability of an evolutionary process model, called the recursive/parallel model, is discussed in Chapter 20.

The first activity performed by the project manager is planning, and one of the early planning tasks is estimation. Recalling the evolutionary process model, planning is revisited after each iteration of the software. Therefore, the plan (and its project estimates) are revisited after each iteration of OOA, OOD, and even OOP.

A key issue that faces a project manager during planning is an estimate of the implemented size of the software. Size is directly proportional to effort and duration. The following OO metrics [LOR94] can provide insight into software size:

Number of scenario scripts (NSS). The number of scenario scripts or use-cases (Chapters 11 and 21) is directly proportional to the number of classes required to meet requirements; the number of states for each class; and the number of methods, attributes, and collaborations. NSS is a strong indicator of program size.

Number of key classes (NKC). A key class focuses directly on the business domain for the problem and will have a lower probability of being implemented via reuse.[9] For this reason, high values for NKC indicate substantial development work. Lorenz and Kidd [LOR94} suggest that between 20 and 40 percent of all classes in a typical OO system are key classes. The remainder support infrastructure (GUI, communications, databases, etc.).

Number of subsystems (NSUB). The number of subsystems provides insight into resource allocation, scheduling (with particular emphasis on parallel development) and overall integration effort.

The metrics NSS, NKC, and NSUB can be collected for past OO projects and are related to the effort expended on the project as a whole and on individual process activities (e.g., OOA, OOD, OOP, and OOT). These data can also be used along with the design metrics discussed earlier in this chapter to compute "productivity metrics" such as average number of classes per developer or average methods per person-month. Collectively, these metrics can be used to estimate effort, duration, staffing, and other information for the current project.

24.8 SUMMARY

Object-oriented software is fundamentally different than software developed using conventional methods. Therefore, the metrics for OO systems focus on measurement that can be applied to the class and the design characteristics—localization, encapsulation, information hiding, inheritance, and object abstraction techniques—that make the class unique.

The CK metrics suite defines six class-oriented software metrics that focus on the class and the class hierarchy. The metrics suite also develops metrics to assess the

9 This will be true only until a robust library of reusable components is developed for a particular domain.

collaborations between classes and the cohesion of methods that reside within a class. At a class-oriented level, the CK metrics suite can be augmented with metrics proposed by Lorenz and Kidd and the MOOD metrics suite. These include measures of class "size" and metrics that provide insight into the degree of specialization for subclasses.

Operation-oriented metrics focus on the size and complexity of individual operations. It is important to note, however, the the primary thrust for OO design metrics is at the class level.

A wide variety of OO metrics have been proposed to assess the testability of an OO system. These metrics focus on encapsulation, inheritance, class complexity, and polymorphism. Many of these metrics have been adapted from the CK metrics suite and metrics proposed by Lorenz and Kidd. Others have been proposed by Binder.

Measurable characteristics of the analysis and design model can assist the project manager for an OO system in planning and tracking activities. The number of scenario scripts (use-cases), key classes, and subsystems provide information about the level of effort required to implement the system.

REFERENCES

[BER95] Berard, E., *Metrics for Object-Oriented Software Engineering,* an Internet posting on comp.software-eng, January 28, 1995.

[BIN94] Binder, R.V., "Object-Oriented Software Testing," *CACM,* vol. 37, no,. 9, September 1994, p. 29.

[CHI94] Chidamber, S.R. and C.F. Kemerer, "A Metrics Suite for Object-Oriented Design," *IEEE Trans. Software Engineering,* vol. SE-20, no. 6, June 1994, pp. 476–493.

[CHI98] Chidamber, S.R., D.P. Darcy, and C.F. Kemerer, "Management Use of Metrics for Object-Oriented Software: An Exploratory Analysis," *IEEE Trans. Software Engineering,* vol. SE-24, no. 8, August 1998, pp. 629–639.

[CHU95] Churcher, N.I. and M.J. Shepperd, "Towards a Conceptual Framework for Object-Oriented Metrics," *ACM Software Engineering Notes,* vol. 20, no. 2, April 1995, pp. 69–76.

[HAR98] Harrison, R., S.J. Counsell, and R.V. Nithi, "An Evaluation of the MOOD Set of Object-Oriented Software Metrics," *IEEE Trans. Software Engineering,* vol. SE-24, no. 6, June 1998, pp. 491–496.

[LOR94] Lorenz, M. and J. Kidd, *Object-Oriented Software Metrics,* Prentice-Hall, 1994.

[WHI97] Whitmire, S., *Object-Oriented Design Measurement,* Wiley, 1997.

[WIL93] Wilde, N. and R. Huitt, "Maintaining Object-Oriented Software," *IEEE Software,* January 1993, pp. 75–80.

[ZUS90] Zuse, H., *Software Complexity: Measures and Methods,* DeGruyter, 1990.

[ZUS97] Zuse, H., *A Framework of Software Measurement,* DeGruyter, 1997.

PROBLEMS AND POINTS TO PONDER

24.1. Review the metrics presented in this chapter and in Chapter 19. How would you characterize the syntactic and semantic differences between metrics for conventional and OO software?

24.2. How does localization affect metrics developed for conventional and OO software?

24.3. Why isn't more emphasis given to OO metrics that address the specific characteristics of operations within a class?

24.4. Review the metrics discussed in this chapter and suggest a few that directly or indirectly address the information hiding design characteristic.

24.5. Review the metrics discussed in this chapter and suggest a few that directly or indirectly address the abstraction design characteristic.

24.6. A class, **X**, has 12 operations. Cyclomatic complexity has been computed for all operations in the OO system and the average value of module complexity is 4. For class **X,** the complexity for operations 1 to 12 is 5, 4, 3, 3, 6, 8, 2, 2, 5, 5, 4, 4, respectively. Compute the weighted methods per class.

24.7. Referring to Figure 20.8, compute the value of DIT for each inheritance tree. What is the value of NOC for the class **X2** for both trees?

24.8. Refer to [CHI94] and present a one-page discussion of the formal definition of the LCOM metric.

24.9. Referring to Figure 20.8B, what is the value of NOA for classes **X3** and **X4**?

24.10. Referring to Figure 20.8B, assume that four operations have been overridden in the inheritance tree (class hierarchy), what is the value of SI for the hierarchy?

24.11. A software team has completed five OO projects to date. The following data have been collected for all size projects:

Project	NSS	NKC	NSUB	Effort (days)
1	34	60	3	900
2	55	75	6	1575
3	122	260	8	4420
4	45	66	2	990
5	80	124	6	2480

A new project is in early stages of OOA. It is estimated that 95 use-cases will be developed for the project. Estimate

a. The total number of classes that will be required to implement the system.

b. The total amount of effort required to implement the system.

24.12. Your instructor will provide you with a list of OO metrics from this chapter. Compute the values of these metrics for one or more of these problems:

a. The design model for the *SafeHome* design.

b. The design model for the PHTRS system described in Problem 12.13.

c. The design model for the video game considered in Problem 22.14.

d. The design model for the e-mail considered in Problem 22.15.

e. The design model for the ATC system considered in Problem 22.16.

FURTHER READINGS AND INFORMATION SOURCES

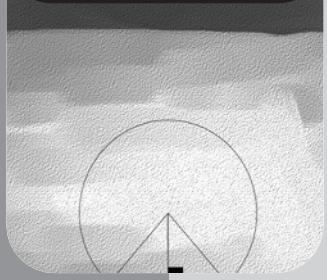
A variety of books on OOA, OOD, and OOT (see Further Readings and Information Sources in Chapters 20, 21, and 22) make passing reference to OO metrics, but few address the subject in any detail. Books by Jacobson (*Object-Oriented Software Engineering,* Addison-Wesley, 1994) and Graham (*Object-Oriented Methods,* Addison-Wesley, 2nd ed., 1993) provide more treatment than most.

Whitmire [WHI97] presents the most comprehensive and mathematically sophisticated treatment of OO metrics published to date. Lorenz and Kidd [LOR94] and Hendersen-Sellers (*Object-Oriented Metrics: Measures of Complexity,* Prentice-Hall, 1996) offer the only other books dedicated to OO metrics. Other books dedicated to conventional software metrics (see Further Readings and Information Sources for Chapters 4 and 19) contain limited discussions of OO metrics.

A wide variety of information sources on object-oriented metrics and related subjects is available on the Internet. An up-to-date list of World Wide Web references that are relevant to OO metrics can be found at the SEPA Web site:
http://www.mhhe.com/engcs/compsci/pressman/resources/OOM.mhtml

PART

Five



ADVANCED TOPICS IN SOFTWARE ENGINEERING

In this part of *Software Engineering: A Practitioner's Approach,* we consider a number of advanced topics that will extend your understanding of software engineering. In the chapters that follow, we address the following questions:

- What notation and mathematical preliminaries ("formal methods") are required to formally specify software?
- What key technical activities are conducted during the cleanroom software engineering process?
- How is component-based software engineering used to create systems from reusable components?
- How does the client/server architecture affect the way in which software is engineered?
- Are software engineering concepts and principles applicable for Web-based applications and products?
- What key technical activities are required for software reengineering?
- What are the architectural options for establishing a CASE tools environment?
- What are the future directions of software engineering?

Once these questions are answered, you'll understand topics that may have a profound impact on software engineering over the next decade.