Analysis is the process of extracting the needs of a system and what the system must do to satisfy the users’ requirements. The goal of object-oriented analysis is to understand the domain of the problem and the system’s responsibilities by understanding how the users use or will use the system. This part consists of Chapters 6, 7, and 8.
Object-Oriented Analysis
Process: Identifying
Use Cases

... just think of all the Christmas pres-
ents that are never removed from their
boxes before being returned.
—Gause and Weinberg [6]

Chapter Objectives
You should be able to define and understand
• The object-oriented analysis process.
• The use-case modeling and analysis.
• Identifying actors.
• Identifying use cases.
• Developing effective documentation.

6.1 INTRODUCTION
The first step in finding an appropriate solution to a given problem is to under-
stand the problem and its domain. The main objective of the analysis is to capture
a complete, unambiguous, and consistent picture of the requirements of the sys-
tem and what the system must do to satisfy the users’ requirements and needs. This
is accomplished by constructing several models of the system that concentrate on
describing what the system does rather than how it does it. Separating the behav-
ior of a system from the way that behavior is implemented requires viewing the
system from the perspective of the user rather than that of the machine.

Analysis is the process of transforming a problem definition from a fuzzy set
of facts and myths into a coherent statement of a system’s requirements. In Chap-
ter 3, we looked at the software development process as three basic transfor-
mations. The objective of this chapter is to describe Transformation 1, which is the
transformation of the users’ needs into a set of problem statements and require-
ments (also known as requirement determination). In this phase of the software
process, you must analyze how the users will use the system and what is needed to accomplish the system’s operational requirements. Analysis involves a great deal of interaction with the people who will be affected by the system, including the actual users and anyone else on whom its creation will have an impact. The analyst has four major tools at his or her disposal for extracting information about a system:

1. Examination of existing system documentation
2. Interviews
3. Questionnaire
4. Observation

In addition, there are minor methods, such as literature review. However, these activities must be directed by a use-case model that can capture the user requirements. The inputs to this phase are the users’ requirements, both written and oral, which will be reduced to the model of the required operational capability of the system.

An object-oriented environment allows the same set of models to be used for analysis, design, and implementation. The analyst is concerned with the uses of the system, identifying the objects and inheritance, and thinks about the events that change the state of objects. The designer adds detail to this model, perhaps designing screens, user interaction, and database access. The thought process flows so naturally from analyst to designer that it may be difficult to tell where analysis ends and design begins [8].

6.2 WHY ANALYSIS IS A DIFFICULT ACTIVITY

Analysis is a creative activity that involves understanding the problem, its associated constraints, and methods of overcoming those constraints. This is an iterative process that goes on until the problem is well understood [11].

Norman [9] explains the three most common sources of requirement difficulties:

1. Fuzzy descriptions
2. Incomplete requirements
3. Unnecessary features

A common problem that leads to requirement ambiguity is a fuzzy and ambiguous description, such as “fast response time” or “very easy and very secure updating mechanisms.” A requirement such as fast response time is open to interpretation, which might lead to user dissatisfaction if the user’s interpretation of a fast response is different from the systems analyst’s interpretation [9].

Incomplete requirements mean that certain requirements necessary for successful system development are not included for a variety of reasons. These reasons could include the users’ forgetting to identify them, high cost, politics within the business, or oversight by the system developer. However, because of the iterative nature of object-oriented analysis and the unified approach (see Chapter 4), most of the incomplete requirements can be identified in subsequent tries.
When addressing features of the system, keep in mind that every additional feature could affect the performance, complexity, stability, maintenance, and support costs of an application. Features implemented by a small extension to the application code do not necessarily have a proportionally small effect on a user interface. For example, if the primary task is selecting a single object, extending it to support selection of multiple objects could make the frequent, simple task more difficult to carry out. A number of other factors also may affect the design of an application. For example, deadlines may require delivering a product to market with a minimal design process, or comparative evaluations may force considering additional features. Remember that additional features and shortcuts can affect the product. There is no simple equation to determine when a design trade-off is appropriate.

Analysis is a difficult activity. You must understand the problem in some application domain and then define a solution that can be implemented with software. Experience often is the best teacher. If the first try reflects the errors of an incomplete understanding of the problems, refine the application and try another run.

### 6.3 BUSINESS OBJECT ANALYSIS: UNDERSTANDING THE BUSINESS LAYER

Business object analysis is a process of understanding the system's requirements and establishing the goals of an application. The main intent of this activity is to understand users' requirements. The outcome of the business object analysis is to identify classes that make up the business layer and the relationships that play a role in achieving system goals.

To understand the users' requirements, we need to find out how they "use" the system. This can be accomplish by developing use cases. Use cases are scenarios for understanding system requirements.

In addition to developing use cases, which will be described in the next section, the uses and the objectives of the application must be discussed with those who are going to use it or be affected by the system. Usually, domain users or experts are the best authorities. Try to understand the expected inputs and desired responses. Defer unimportant details until later. State what must be done, not how it should be done. This, of course, is easier said than done. Yet another tool that can be very useful for understanding users' requirements is preparing a prototype of the user interface. Preparation of a prototype usually can help you better understand how the system will be used, and therefore it is a valuable tool during business object analysis. (We defer the discussion of prototyping a user interface to Chapter 12.)

Having established what users want by developing use cases then documenting and modeling the application, we can proceed to the design and implementation. The unified approach (UA) steps can overlap each other. The process is iterative, and you may have to backtrack to previously completed steps for another try. Separating the what from the how is no simple process. Fully understanding a problem and defining how to implement it may require several tries or iterations. In this chapter, we see how a use-case model can assist us in capturing an application's requirements.
6.4 USE-CASE DRIVEN OBJECT-ORIENTED ANALYSIS: THE UNIFIED APPROACH

The object-oriented analysis (OOA) phase of the unified approach uses actors and use cases to describe the system from the users' perspective. The actors are external factors that interact with the system; use cases are scenarios that describe how actors use the system. The use cases identified here will be involved throughout the development process.

The OOA process consists of the following steps (see Figure 6–1):

1. Identify the actors:
   • Who is using the system?
   • Or, in the case of a new system, who will be using the system?
2. Develop a simple business process model using UML activity diagram.
3. Develop the use case:
   • What are the users doing with the system?
   • Or, in case of the new system, what will users be doing with the system?
   • Use cases provide us with comprehensive documentation of the system under study.
4. Prepare interaction diagrams:
   • Determine the sequence.
   • Develop collaboration diagrams.
5. Classification—develop a static UML class diagram:
   • Identify classes.
   • Identify relationships.
   • Identify attributes.
   • Identify methods.
6. Iterate and refine: If needed, repeat the preceding steps.

This chapter focuses on steps 1 to 3.

**FIGURE 6–1**
The object-oriented analysis process in the Unified Approach (UA).
6.5 BUSINESS PROCESS MODELING

This is not necessarily the start of every project, but when required, business processes and user requirements may be modeled and recorded to any level of detail. This may include modeling as-is processes and the applications that support them and any number of phased, would-be models of reengineered processes or implementation of the system. These activities would be enhanced and supported by using an activity diagram. Business process modeling can be very time consuming, so the main idea should be to get a basic model without spending too much time on the process. The advantage of developing a business process model is that it makes you more familiar with the system and therefore the user requirements and also aids in developing use cases. For example, let us define the steps or activities involved in using your school library. These activities can be represented with an activity diagram (see Figure 6–2).

Developing an activity diagram of the business process can give us a better understanding of what sort of activities are performed in a library by a library member.

6.6 USE-CASE MODEL

Use cases are scenarios for understanding system requirements. A use-case model can be instrumental in project development, planning, and documentation of systems

![Activity Diagram](Image)
requirements. A use case is an interaction between users and a system; it captures the goal of the users and the responsibility of the system to its users. For example, take a car; typical uses of a car include "take you different places" or "haul your stuff" or a user may want to use it "off the road." The use-case model describes the uses of the system and shows the courses of events that can be performed. In other words, it shows a system in terms of its users and how it is being used from a user point of view. Furthermore, it defines what happens in the system when the use case is performed. In essence, the use-case model tries to systematically identify uses of the system and therefore the system's responsibilities. A use-case model also can discover classes and the relationships among subsystems of the systems.

A use-case model can be developed by talking to typical users and discussing the various things they might want to do with the application being prepared. Each use or scenario represents what the user wants to do. Each use case must have a name and short textual description, no more than a few paragraphs (see Chapter 4).

Since the use-case model provides an external view of a system or application, it is directed primarily toward the users or the "actors" of the systems, not its implementers (see Figure 6-3). The use-case model expresses what the business or application will do and not how; that is the responsibility of the UML class dia-

**FIGURE 6-3**

Some uses of a library. As you can see, these are uses of external views of the library system by an actor such as a member, circulation clerk, or supplier instead of a developer of the library system. The simpler the use-case model, the more effective it will be. It is not wise to capture all the details right at the start; you can do that later.
gram [7]. The UML class diagram, also called an object model, represents the static relationships between objects, inheritance, association, and the like. The object model represents an internal view of the system, as opposed to the use-case model, which represents the external view of the system. The object model shows how the business is run. Jacobson, Ericsson, and Jacobson call the use-case model a “what model,” in contrast to the object model, which is a “how model.”

6.6.1 Use Cases under the Microscope

An important issue that can assist us in building correct use cases is the differentiation between user goals and system interactions [5]. Use cases represent the things that the user is doing with the system, which can be different from the users’ goals. However, by focusing on users’ goals first, we can come up with use cases to satisfy them. Let us take a closer look at the definition of use case by Jacobson et al. [7, italics added to highlight the words that are discussed next]: “A Use Case is a sequence of transactions in a system whose task is to yield results of measurable value to an individual actor of the system.”

Now let us take a look at the key words of this definition:

- **Use case.** Use case is a special flow of events through the system. By definition, many courses of events are possible and many of these are very similar. It is suggested that, to make a use-case model meaningful, we must group the courses of events and call each group a use-case class. For example, how you would borrow a book from the library depends on whether the book is located in the library, whether you are the member of the library, and so on. All these alternatives often are best grouped into one or two use cases, called Borrow books and Get an interlibrary loan (we will look at the relationships of these two use cases in the next section). By grouping the uses cases, we can manage complexities and reduce the number of use cases in a package.

- **Actors.** An actor is a user playing a role with respect to the system. When dealing with actors, it is important to think about roles rather than just people and their job titles [5]. For instance, a first-class passenger may play the role of business-class passenger. The actor is the key to finding the correct use cases. Actors carry out the use cases. A single actor may perform many use cases; furthermore, a use case may have several actors performing it. An actor also can be an external system that needs some information from the current system. Actors can be the ones that get value from the use case, or they can just participate in the use case [5].

- **In a system.** This simply means that the actors communicate with the system’s use case.

- **A measurable value.** A use case must help the actor to perform a task that has some identifiable value; for example, the performance of a use case in terms of price or cost. For example, borrowing books is something of value for a member of the library.

---

1The how model here does not mean how the system can be implemented but how the scenarios can be handled internally.
FIGURE 6-4
The use-case diagram depicts the extends and uses relationships, where the interlibrary loan is a special case of checking out books. Entering into the system is common to get an interlibrary loan, borrow books, and return books use cases, so it is being "used" by all these use cases.

- **Transaction.** A transaction is an atomic set of activities that are performed either fully or not at all. A transaction is triggered by a stimulus from an actor to the system or by a point in time being reached in the system.

The following are some examples of use cases for the library (see Figure 6–4). Three actors appear in Figure 6–4: a member, a circulation clerk, and a supplier.

- **Use-case name: Borrow books.** A member takes books from the library to read at home, registering them at the checkout desk so the library can keep track of its books. Depending on the member’s record, different courses of events will follow.
- **Use-case name: Get an interlibrary loan.** A member requests a book that the library does not have. The book is located at another library and ordered through an interlibrary loan.
- **Use-case name: Return books.** A member brings borrowed books back to the library.
- **Use-case name: Check library card.** A member submits his or her library card to the clerk, who checks the borrower’s record.
- **Use-case name: Do research.** A member comes to the library to do research. The member can search in a variety of ways (such as through books, journals, CD-ROM, WWW) to find information on the subjects of that research.
- Use-case name: Read books, newspaper. A member comes to the library for a quiet place to study or read a newspaper, journal, or book.
- Use-case name: Purchase supplies. The supplier provides the books, journals, and newspapers purchased by the library.

In Figure 6–4, the library has an environment with three types of actors (member, circulation clerk, and supplier) and seven use cases (borrow books, return books, get an interlibrary loan, do research, read books or newspaper, and purchase supplies).

6.6.2 Uses and Extends Associations

A use-case description can be difficult to understand if it contains too many alternatives or exceptional flows of events that are performed only if certain conditions are met as the use-case instance is carried out [7]. A way to simplify the description is to take advantage of extends and uses associations. The extends and uses associations often are sources of confusion, so let us take a look at these relationships.

The *extends association* is used when you have one use case that is similar to another use case but does a bit more or is more specialized; in essence, it is like a subclass. In our example, *checking out a book* is the basic use case. This is the case that will represent what happens when all goes smoothly. However, many things can affect the flow of events. For example, the book already might be checked out or the library might not have the requested book. Therefore, we cannot always perform the usual behavior associated with the given use case and need to create other use cases to handle the new situations. Of course, one option is to put this variation within the use case. However, the use case quickly would become cluttered with lots of special logic, which would obscure the normal flow [5].

To remedy this problem, we can use the extends association. Here, you put the base or normal behavior in one use case and the unusual behaviors somewhere else; but instead of cutting and pasting the shared behavior between the base (common) and more specialized use cases, you utilize an extends association to expand the common behavior to fit the special circumstances. Figure 6–4 “extends” Figure 6–3 to include extends and uses associations.

The *uses association* occurs when you are describing your use cases and notice that some of them have subflows in common. To avoid describing a subflow more than once in several use cases, you can extract the common subflow and make it a use case of its own. This new use case then can be used by other use cases. The relationships among the other use cases and this new extracted use case is called a *uses association*. The uses association helps us avoid redundancy by allowing a use case to be shared. For example, checking a library card is common among the borrow books, return books, and interlibrary loan use cases (see Figure 6–4).

The similarity between extends and uses associations is that both can be viewed as a kind of inheritance. When you want to share common sequences in several use cases, utilize the *uses association* by extracting common sequences into a new, shared use case. The *extends association* is found when you add a bit more specialized, new use case that extends some of the use cases that you have.

Use cases could be viewed as concrete or abstract. An *abstract use case* is not complete and has no initiation actors but is used by a *concrete use case*, which
does interact with actors. This inheritance could be used at several levels. Abstract
use cases also are the use cases that have uses or extends associations. All the use
cases depicted in Figure 6–4 are concrete, since they all have initiation actors.

Fowler and Scott provide us excellent guidelines for addressing variations in
use-case modeling [5]:

1. Capture the simple and normal use case first.
2. For every step in that use case, ask
   • What could go wrong here?
   • How might this work out differently?
3. Extract common sequences into a new, shared use case with the uses associa-
tion. If you are adding more specialized or exceptional uses cases, take advan-
tage of use cases you already have with the extends association.

6.6.3 Identifying the Actors

Identifying the actors is (at least) as important as identifying classes, structures,
associations, attributes, and behavior. The term actor represents the role a user
plays with respect to the system. When dealing with actors, it is important to think
about roles rather than people or job titles [5]. A user may play more than one role.
For instance, a member of a public library also may play the role of volunteer at
the help desk in the library. However, an actor should represent a single user; in
the library example, the member can perform tasks some of which can be done by
others and others that are unique. However, try to isolate the roles that the users
can play [1]. (See Figure 6–5.)

You have to identify the actors and understand how they will use and interact
with the system. In a thought-provoking book on requirement analysis, Gause and
Weinberg [6, pp. 69–70] explain what is known as the railroad paradox:

FIGURE 6–5
The difference between users and actors.

![Diagram showing relationships between users, actors, and use cases.]

USER  |  Can play the role of  |  ACTOR  |  Performs  |  USE CASE
---|---|---|---|---
Sylvia | Member |  | Borrow book |
| Jackie | Employee |  | Order books |
| Lili | Volunteer |  | Check IDs |
When trying to find all users, we need to beware of the Railroad Paradox. When railroads were asked to establish new stops on the schedule, they “studied the requirements,” by sending someone to the station at the designated time to see if anyone was waiting for a train. Of course, nobody was there because no stop was scheduled, so the railroad turned down the request because there was no demand.

Gause and Weinberg concluded that the railroad paradox appears everywhere there are products and goes like this (which should be avoided):

1. The product is not satisfying the users.
2. Since the product is not satisfactorily, potential users will not use it.
3. Potential users ask for a better product.
4. Because the potential users do not use the product, the request is denied.

Therefore, since the product does not meet the needs of some users, they are not identified as potential users of a better product. They are not consulted and the product stays bad [6]. The railroad paradox suggests that a new product actually can create users where none existed before. Candidates for actors can be found through the answers to the following questions:

- Who is using the system? Or, who is affected by the system? Or, which groups need help from the system to perform a task?
- Who affects the system? Or, which user groups are needed by the system to perform its functions? These functions can be both main functions and secondary functions, such as administration.
- Which external hardware or other systems (if any) use the system to perform tasks?
- What problems does this application solve (that is, for whom)?
- And, finally, how do users use the system (use case)? What are they doing with the system.

When requirements for new applications are modeled and designed by a group that excludes the targeted users, not only will the application not meet the users’ needs, but potential users will feel no involvement in the process and not be committed to giving the application a good try. Always remember Veblen’s principle: “There’s no change, no matter how awful, that won’t benefit some people; and no change, no matter how good, that won’t hurt some.”

Another issue worth mentioning is that actors need not be human, although actors are represented as stick figures within a use-case diagram. An actor also can be an external system. For example, an accounting system that needs information from a system to update its accounts is an actor in that system [5].

Jacobson et al. provide us with what I call the two-three rule for identifying actors: Start with naming at least two, preferably three, people who could serve as the actors in the system. Other actors can be identified in the subsequent iterations. Remember this, like any other software development process, is an iterative process. For example, assume we are modeling a company that specializes in marketing jewelry. The first actor that comes to mind is the final customer; actually three different regular customers would buy the product. Another type of actor is the jewelry buyers for exclusive stores; they know all about quality and nothing
else. A third type of customer is boutique owners, who know what designs are in fashion. Each of these individuals requires his or her own use case, since they represent different roles that can be played in the system. Still other actors might be identified through subsequent iterations.

6.6.4 Guidelines for Finding Use Cases

When you have defined a set of actors, it is time to describe the way they interact with the system. This should be carried out sequentially, but an iterated approach may be necessary. Here are the steps for finding use cases [1]:

1. For each actor, find the tasks and functions that the actor should be able to perform or that the system needs the actor to perform. The use case should represent a course of events that leads to a clear goal (or, in some cases, several distinct goals that could be alternatives for the actor or for the system).
2. Name the use cases (see Section 6.6.8).
3. Describe the use cases briefly by applying terms with which the user is familiar. This makes the description less ambiguous.

Once you have identified the use-case candidates, it may not be apparent that all of these use cases need to be described separately; some may be modeled as variants of others. Consider what the actors want to do.

It is important to separate actors from users. The actors each represent a role that one or several users can play. Therefore, it is not necessary to model different actors that can perform the same use case in the same way. The approach should allow different users to be different actors and play one role when performing a particular actor’s use case. Thus, each use case has only one main actor. To achieve this, you have to

- Isolate users from actors.
- Isolate actors from other actors (separate the responsibilities of each actor).
- Isolate use cases that have different initiating actors and slightly different behavior (if the actor had been the same, this would be modeled by a use-case alternative behavior) [1].

While finding use cases, you might have to make changes to your set of actors. All actor changes should be updated in the textual description of actors and use cases. The change should be carried out with care, since changes to the set of actors affect the use cases as well.

When specifying use cases, you might discover that some of them are variants of each other. If so, try to see how you can reuse the use case through extends or uses associations [1].

6.6.5 How Detailed Must a Use Case Be? When to Stop Decomposing and When to Continue

A use case, as already explained, describes the courses of events that will be carried out by the system. Jacobson et al. believe that, in most cases, too much detail may not be very useful.
During analysis of a business system, you can develop one use-case diagram as the system use case and draw packages on this use case to represent the various business domains of the system. For each package, you may create a child use-case diagram (see the case in Section 6.7 for an example). On each child use-case diagram, you can draw all of the use cases of the domain, with actions and interactions. You can further refine the way the use cases are categorized. The extends and uses relationships can be used to eliminate redundant modeling of scenarios.

When should use cases be employed? Use cases are an essential tool in capturing requirements and planning and controlling any software development project. Capturing use cases is a primary task of the analysis phase. Although most use cases are captured at the beginning of the project, you will uncover more as you proceed.

How many use cases do you need? Ivar Jacobson believes that, for a 10-person-year project, he would expect 20 use cases (not counting the uses and extends associations). Other researchers, such as Fowler and Scott, would come up with 100 use cases for a project of the same magnitude. Some prefer smaller grained, more detailed use cases. There is no magic formula; you need to be flexible and work with whatever magnitude you find comfortable [5]. The UML specification recommends that at least one scenario be prepared for each significantly different kind of use case instance. Each scenario shows a different sequence of interactions between actors and the system, with all decisions definite. When you have arrived at the lowest use-case level, which cannot be broken down any further, you may create a sequence diagram and an accompanying collaboration diagram for the use case. With the sequence and collaboration diagrams, you can model the implementation of the scenario [10].

6.6.6 Dividing Use Cases into Packages

Each use case represents a particular scenario in the system. You may model either how the system currently works or how you want it to work. Typically, a design is broken down into packages. You must narrow the focus of the scenarios in your system. For example, in a library system, the various scenarios involve a supplier providing books or a member doing research or borrowing books. In this case, there should be three separate packages, one each for Borrow books, Do research, and Purchase books. Many applications may be associated with the library system and one or more databases used to store the information (see Figure 6–6).

6.6.7 Naming a Use Case

Use-case names should provide a general description of the use-case function. The name should express what happens when an instance of the use case is performed. Jacobson et al. recommend that the name should be active, often expressed in the form of a verb (Borrow) or verb and noun (Borrow books). The naming should be done with care; the description of the use case should be descriptive and consistent. For example, the use case that describes what happens when a person deposits money into an ATM machine could be named either receive money or deposit money.
FIGURE 6-6
A library system can be divided into many packages, each of which encompasses multiple use cases.

6.7 DEVELOPING EFFECTIVE DOCUMENTATION

Documenting your project not only provides a valuable reference point and form of communication but often helps reveal issues and gaps in the analysis and design. A document can serve as a communication vehicle among the project's team members, or it can serve as an initial understanding of the requirements. Blum [3] concludes that management has responsibility for resources such as software, hardware, and operational expenses. In many projects, documentation can be an important factor in making a decision about committing resources. Application software is expected to provide a solution to a problem. It is very difficult, if not impossible, to document a poorly understood problem. The main issue in documentation during the analysis phase is to determine what the system must do. Decisions about how the system works are delayed to the design phase. Blum raises the following questions for determining the importance of documentation: How will a document be used? (If it will not be used, it is not necessary.) What is the objective of the document? What is the management view of the document? Who are the readers of the document?
6.7.1 Organization Conventions for Documentation

The documentation depends on the organization’s rules and regulations. Most organizations have established standards or conventions for developing documentation. However, in many organizations, the standards border on the nonexistent. In other cases, the standards may be excessive. Too little documentation invites disaster; too much documentation, as Blum put it, transfers energy from the problem-solving tasks to a mechanical and unrewarding activity. Each organization determines what is best for it, and you must respond to that definition and refinement [3]. Bell and Evans [2] provide us with guidelines and a template for preparing a document that has been adapted for documenting the unified approach’s systems development (see Appendix A). Remember that your modeling effort becomes the analysis, design, and testing documentation. However, this template which is based on the unified approach life cycle (see Figure 1–1) assists you in organizing and composing your models into an effective documentation.

6.7.2 Guidelines for Developing Effective Documentation

Bell and Evans [2] provide us the following guidelines for making documents fit the needs and expectations of your audience:

- **Common cover.** All documents should share a common cover sheet that identifies the document, the current version, and the individual responsible for the content. As the document proceeds through the life cycle phases, the responsible individual may change. That change must be reflected in the cover sheet [2]. Figure 6–7 depicts a cover sheet template.

- **80–20 rule.** As for many applications, the 80–20 rule generally applies for documentation: 80 percent of the work can be done with 20 percent of the documentation. The trick is to make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those (few) who need to know.

- **Familiar vocabulary.** The formality of a document will depend on how it is used and who will read it. When developing a documentation use a vocabulary that your readers understand and are comfortable with. The main objective here is to communicate with readers and not impress them with buzz words.

- **Make the document as short as possible.** Assume that you are developing a manual. The key in developing an effective manual is to eliminate all repetition; present summaries, reviews, organization chapters in less than three pages; and make chapter headings task oriented so that the table of contents also could serve as an index [4].

- **Organize the document.** Use the rules of good organization (such as the organization’s standards, college handbooks, Strunk and White’s *Elements of Style*, or the University of Chicago Manual of Style) within each section. Appendix A provides a template for developing documentation for a project. Most CASE tools provide documentation capability by providing customizable reports. The purpose of these guidelines is to assist you in creating an effective documentation.
6.8 CASE STUDY: ANALYZING THE VIANET BANK ATM—THE USE-CASE DRIVEN PROCESS

6.8.1 Background

Much of the work that must be done in the early stages of the system development process involves gathering requirements and other related information. These activities focus on gaining a better understanding of the business problem to be solved and the requirements and restrictions related to the application being developed. As explained in the previous section, use cases are employed to capture information on how a system or business currently works or how you wish it to work. Once you have a good understanding of the requirements, you can analyze it further and begin to design your application. Using a CASE tool, such as the Popkin Object Architect or a similar tool, enables you to systematically capture requirements, identify classes, design, and finally implement the application.

Let us review object-oriented analysis, which is divided into the following activities:

1. Identify the actors: Who is using the system?
2. Develop a business process model using a UML activity diagram.
3. Develop the use case: What are the users doing with the system?
4. Develop interaction diagrams.
5. Develop a static UML class diagram.
6. If needed, repeat the preceding steps.

The following section provides a description of the ViaNet bank ATM system’s requirements.

- The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the touch screen at the ViaNet bank ATM kiosk. Each transaction must be recorded, and the client must be able to review all transactions performed against a given account. Recorded transactions must include the date, time, transaction type, amount, and account balance after the transaction.
* A ViaNet bank client can have two types of accounts: a checking account and savings account. For each checking account, one related savings account can exist.
* Access to the ViaNet bank accounts is provided by a PIN code consisting of four integer digits between 0 and 9.
* One PIN code allows access to all accounts held by a bank client.
* No receipts will be provided for any account transactions.
* The bank application operates for a single banking institution only.
* Neither a checking nor a savings account can have a negative balance. The system should automatically withdraw money from a related savings account if the requested withdrawal amount on the checking account is more than its current balance. If the balance on a savings account is less than the withdrawal amount requested, the transaction will stop and the bank client will be notified.

In this chapter, we identify the actors and use cases of the ViaNet bank ATM system that will be used by subsequent chapters.

### 6.8.2 Identifying Actors and Use Cases for the ViaNet Bank ATM System

The bank application will be used by one category of users: bank clients. Notice that identifying the actors of the system is an iterative process and can be modified as you learn more about the system. The actor of the bank system is the bank client. The bank client must be able to deposit an amount to and withdraw an amount from his or her accounts using the bank application.

The following scenarios show use-case interactions between the actor (bank client) and the bank. In real-life application these use cases are created by system requirements, examination of existing system documentation, interviews, questionnaire, observation, etc.

* Use-case name: *Bank ATM transaction*. The bank clients interact with the bank system by going through the approval process. After the approval process, the bank client can perform the transaction. Here are the steps in the ATM transaction use case:
  1. Insert ATM card.
  2. Perform the approval process.
  3. Ask type of transaction.
  4. Enter type of transaction.
  5. Perform transaction.
  6. Eject card.
  7. Request take card.
  8. Take card.

These steps are shown in the Figure 6–8 activity diagram.

* Use-case name: *Approval process*. The client enters a PIN code that consists of four digits. If the PIN code is valid, the client’s accounts become available. (See Figure 6–9.) Here are the steps:
  1. Request password.
2. Enter password.
3. Verify password.

- Use-case name: *Invalid PIN*. If the PIN code is not valid, an appropriate message is displayed to the client. This use case extends the approval process. (See Figure 6-9.)

- Use-case name: *Deposit amount*. The bank clients interact with the bank system after the approval process by requesting to deposit money to an account. The client selects the account for which a deposit is going to be made and enters an amount in dollar currency. The system creates a record of the transaction. (See Figure 6-10.) This use case extends the bank ATM transaction use case. Here are the steps:
  1. Request account type.
  2. Request deposit amount.
  3. Enter deposit amount.
  4. Put the check or cash in the envelope and insert it into ATM.

- Use-case name: *Deposit savings*. The client selects the savings account for which a deposit is going to be made. All other steps are similar to the deposit amount use case. The system creates a record of the transaction. This use case extends the deposit amount use case. (See Figure 6-11.)
• Use-case name: Deposit checking. The client selects the checking account for which a deposit is going to be made. All other steps are similar to the deposit amount use case. The system creates a record of the transaction. This use case extends the deposit amount use case. (See Figure 6-10.)

• Use-case name: Withdraw amount. The bank clients interact with the bank system (after the approval process) by requesting to withdraw money from an account. The client tries to withdraw an amount from a checking account. After verifying that the funds are sufficient, the transaction is performed. The system creates a record of the transaction. This use case extends the bank ATM transaction use case. (See Figure 6-10.) Here are the steps:
  1. Request account type.
  2. Request withdrawal amount.
  3. Enter withdrawal amount.
  4. Verify sufficient funds.
  5. Eject cash.

• Use-case name: Withdraw checking. The client tries to withdraw an amount from his or her checking account. The amount is less than or equal to the checking account's balance, and the transaction is performed. The system creates a record of the transaction. This use case extends the withdraw amount use case. (See Figure 6-10.)
• Use-case name: *Withdraw more from checking*. The client tries to withdraw an amount from his or her checking account. If the amount is more than the checking account’s balance, the insufficient amount is withdrawn from the related savings account. The system creates a record of the transaction and the withdrawal is successful. This use case extends the withdraw checking use case and uses the withdraw savings use case. (See Figure 6–10.)

• Use-case name: *Withdraw savings*. The client tries to withdraw an amount from a savings account. The amount is less than or equal to the balance and the transaction is performed on the savings account. The system creates a record of the transaction since the withdrawal is successful. This use case extends the withdraw amount use case. (See Figure 6–11.)

• Use-case name: *Withdraw savings denied*. The client withdraws an amount from a savings account. If the amount is more than the balance, the transaction is halted and a message is displayed. This use case extends the withdrawn savings use case. (See Figure 6–11.)

• Use-case name: *Checking transaction history*. The bank client requests a history of transactions for a checking account. The system displays the transaction his-
The savings account use-cases package.

The use-case list contains at least one scenario of each significantly different kind of use-case instance. Each scenario shows a different sequence of interactions between actors and the system, with all decisions definite. If the scenario consists of an if statement, for each condition create one scenario.

Note that the extends association is used when you have a use case that is similar to another use case but does a bit more. In essence, it is a subclass. In the example, the Checking withdraw use case extends the Withdraw amount use case. The Withdraw amount use case represents the case when all goes smoothly. However, many things can affect the flow of events, such as when the withdrawal is for more than the amount of money in the checking account. Withdraw more from checking is the use case that extends the Checking withdraw. You can put this variation within the Checking withdraw use case, too. However, this would clutter the use case with lots of special logic, which would obscure the normal flow. To review, the uses association occurs when a behavior is common to more than one
use case and you want to avoid copying the description of that behavior. The Approval process is such a use case that is used by Bank transaction use case.

As you can see, use cases are an essential tool for identifying requirements. Developing use cases is an iterative process. Although most use cases are generated at this phase of system development, you will uncover more as you proceed. Fowler and Scott advise us to keep an eye out for them at all times. Every use case represents a potential requirement.

### 6.8.3 The ViaNet Bank ATM Systems' Packages

Each use case represents a particular scenario in the system. As explained earlier, it is better to break down the use cases into packages. Narrow the focus of the scenarios in the system. In the bank system, the various scenarios involve checking account, savings account, and general bank transactions. (See Figure 6–12.)

Remember, use case is a method for capturing the way a system or business works. Use cases are used to model the scenarios. The scenarios are described textually or through a sequence of steps. Modeling with use cases is a recommended tool in finding the objects of a system. In the next chapter, we look at identifying classes based on the use cases identified here.

### 6.9 SUMMARY

This chapter provides a detailed discussion of use-case driven object-oriented analysis process and how to develop use cases. The main objective of the analysis is to capture a complete, unambiguous, and consistent picture of the requirements of the system. This is accomplished by constructing several models of the system. These models concentrate on describing what the system does rather than how the system does it. Separating the behavior of a system from the way it is implemented requires viewing the system from the perspective of the users rather than that of the machine. Analysis is a creative activity that involves understanding the prob-
lem, its associated constraints, and methods of overcoming those constraints. This is an iterative process that goes on until the problem is well understood. The main objective of object-oriented analysis is to find out what the problem is by developing a use-case model, which Jacobson et al. call the “what model.”

We saw that use cases are an essential tool in capturing requirements. Capturing use cases is one of the first things to do in coming up with requirements. Every use case is a potential requirement. A use-case model can be developed by talking to typical users and discussing the various things they might want to do with the application. Each use case or scenario represents what the user wants to do. Each use case must have a name and short textual description, no more than a few paragraphs.

Requirements must be traceable across analysis, design, coding, and testing. The unified approach follows Jacobson et al.’s life cycle to produce systems that can be traced across all phases of the developments.

The key in developing effective documentation is to eliminate all repetition; present summaries, reviews, organization chapters in less than three pages; and make chapter headings task oriented so that the table of contents also could serve as an index.

Use the 80–20 rule: 80 percent of the work can be done with 20 percent of the documentation. Make sure that the 20 percent is easily accessible and the rest (80 percent) is available to those few who need to know.

Appendix A provides a template for documentation. However, for the most part, the modeling activity is the main source of documenting the OOA.

**KEY TERMS**

Abstract use case (p. 133)
Actor (p. 128)
Concrete use case (p. 133)
Extends association (p. 133)
Two-three rule (p. 135)
Use cases (p. 128)
Use-case model (p. 130)
Uses association (p. 133)

**REVIEW QUESTIONS**

1. What is the purpose of analysis? Why do we need analysis?
2. Why is analysis a difficult task?
3. What approach is proposed in this chapter to manage complexity in the analysis phase?
4. What is the what model?
5. What is a use-case model?
6. What is involved in the analysis process? Where should we start?
7. Describe the basic activities in object-oriented analysis.
8. Why is use-case modeling useful in analysis?
9. Who are the actors?
10. Why are uses and extends associations useful in use-case modeling?
11. How would you identify actors?
12. What is the 80–20 rule?

13. Why is documentation an important part of analysis?

14. The criterion for something to be an actor is that it lies outside the part of the business being modeled; yet it interacts with that part in some way. Why must the actor be someone or something outside the part of the business being modeled?

15. What is the difference between users and actors?

PROBLEMS

1. Lee Turner is director of information systems (IS) for the city of Providence. The IS department's customers are the public library, the fire department, the police department, the finance department, the sanitation department, and the water department. Lee believes close communication with these customers is the key to meeting their needs. Currently, the police and fire departments need fast access to a map of the city for dispatching the city's ambulance and fire trucks to accident sites.

   a. Who are the actors?
   
   b. How would you incorporate the users' needs into the system development process?
   
   c. Develop a simple use-case model.

2. The Book Store sells textbooks but also many other items, ranging from Rhode Island College (RIC) sweatshirts to computers. The text purchasing department has unique characteristics, including advance notice from faculty members and issues dealing with unsold copies. Purchasing the other items is as for any retail store. An extension of both areas is the checkout (or sales) process. This process should include the cash registers, scanners, and sales slips. In fact, this process often is unduly slow. Develop an activity diagram to show the business process of the book store.

3. All students must provide information about inoculations and other health information when entering RIC. A physical exam may be necessary at the Health Services office. This health information is recorded on a health card, which often is hard to file and retrieve when needed, due to the number of cards. Delays in filling and retrieving cause problems for other functions on campus. For example, a student may not be enrolled unless the card is filed. What if the card is filed but not retrieved or otherwise recorded? A student needs the data recorded to participate in organized athletics. Again, the difficulty in retrieving the card can produce difficulties for the Athletics department. How would you go about creating use cases? Come up with one or two use-cases for health services.

4. Grandma, the owner of Grandma's Soup Unlimited, cans her soup and sells it by mail order. She wants to redo her ordering and shipping and, down the road, she would also like to have a new payroll system in place. Your job is to design a system to handle ordering and shipping with an eye on the payroll system.

   She, along with most of her employees (majority of them are college students), likes the idea of the Graphical User interface interfaces and down the road utilizing e-commerce (electronic commerce) or doing business on the Internet. Grandma states that she would like this new system to be visual and easy to use and assumes it will be able to do all the things her spreadsheet program did.

   During the interview, Grandma mentioned that she wants her system to be "expandable." In other words, she wants it to be able to grow with her needs. By grow, she was also referring to her store because she plans on opening a few more stores in the near future. She wants the flexibility of adding new/different types of services, such as new soups, cans, and even hiring different types of employees. Grandma is not sure what these different employees will do yet, but she assures us there will be some changes in the future. She would also like to have better reports made from her data.
Some other aspects of Grandma's payroll system were also discussed during the interview. There are many different kinds of employees, such as clerk, telephone operator, cook, manager, even owner of future stores. Some of these employees are full-time, others are part-time, and some are even salaried. All of the employees have some commonalities, but there are also many differences among them.

Grandma makes a variety of soups, including Beef-Barley Soup, Beef Stew, Cheese Soup, Cheesy Chicken-Corn Chowder, Chicken Vegetable-Noodle Soup, Cream of Broccoli Soup, Cream of Chicken Soup, Cream of Potato Soup, Cream of Onion Soup, Cream of Pea Soup, Fish Chowder, French Onion Soup, New England Clam Chowder, and Old Fashioned Vegetable-Beef Soup. Each uses different ingredients. For instance, French Onion Soup contains onion, butter, flour, French bread, cheddar cheese, wine, and vegetables.

Here are some more requirements:

- Grandma's philosophy about customer service is "first come, first served."
- Grandma needs a list of pending orders. (Keep in mind that sometimes the orders really pile up.)
- Develop some of the use cases of the Grandma Soup Unlimited system.

REFERENCES

Chapter Objectives
You should be able to define and understand
• The concept of classification.
• How to identify classes with the noun phrase approach.
• How to identify classes with the common class patterns approach.
• How to identify classes and object behavior analyzed by sequence/collaboration modeling.
• How to identify classes with the classes, responsibilities, and collaborators (CRC) approach.

7.1 INTRODUCTION
Object-oriented analysis is a process by which we can identify classes that play a role in achieving system goals and requirements. Unfortunately, classes rarely just are there for the picking [3]. Identification of classes is the hardest part of object-oriented analysis and design. Booch [2, p. 145] argues that, “There is no such a thing as the perfect class structure, nor the right set of objects. As in any engineering discipline, our design choice is compromisingly shaped by many competing factors.” Gabriel, White, and Bobrow, the three designers of the object-oriented language CLOS, respond to the issue of how to identify classes: “That’s a fundamental question for which there is no easy answer. I try things” [4].
In this chapter, we look at four approaches for identifying classes and their behaviors in the problem domain. Discovering classes is one of the hardest activities in object-oriented analysis. However, the process is incremental and iterative and, furthermore, as you gain experience it will become easier to identify the classes.

### 7.2 CLASSIFICATIONS THEORY

**Classification**, the process of checking to see if an object belongs to a category or a class, is regarded as a basic attribute of human nature.

Booch [2, p. 146] explains that,

intelligent classification is part of all good science. . . . Classification guides us in making decisions about modularization. We may choose to place certain classes and objects together in the same module or in different modules, depending upon the sameness we find among these declarations; coupling and cohesion are simply measures of this same-ness. Classification also plays a role in allocating processes to procedures. We place certain processes together in the same processor or different processors, depending upon packaging, performance, or reliability concerns.

Human beings classify information every instant of their waking lives. We recognize the objects around us, and we move and act in relation to them. A human being is a very sophisticated information system, partly because he or she possesses a superior classification capability [11]. For example, when you see a new model of a car, you have no trouble identifying it as a car. What has occurred here, even though you may never have seen this particular car before, is that you not only can immediately identify it as a car, but you also can guess the manufacturer and model. Clearly, you have some general idea of what cars look like, sound like, do, and are good for—you have a notion of car-kind or, in object-oriented terms, the class car.

Classes are an important mechanism for classifying objects. The chief role of a class is to define the attributes, methods, and applicability of its instances. The class car, for example, defines the property color. Each individual car (formally, each instance of the class car) will have a value for this property, such as maroon, yellow, or white.

It is fairly natural to partition the world into objects that have properties (attributes) and methods (behaviors). It is common and useful partitioning or classification, but we also routinely divide the world along a second dimension: We distinguish classes from instances. A class is a specification of structure, behavior, and the description of an object. Classification is concerned more with identifying the class of an object than the individual objects within a system. Martin and Odell explain that classes are important because they create conceptual building blocks for designing systems:

In object-oriented programming, these building blocks guide the designer in defining the classes and their data structures. In addition, object types (classes) provide an index
CHAPTER 7: OBJECT ANALYSIS: CLASSIFICATION

FIGURE 7-1
The same object, Betty, can be classified in many ways.

for system process. For instance, operations such as Hire, Promote, Retire, and Fire are intimately tied to the object type (class) Employee, because they change the state of an employee. In other words, an object should only be manipulated via the operations associated with its type. Without object types (classes), then, operations cannot be defined properly. [6, p. 76]

Stuei and Gonzalez describe the recognition of concrete patterns or classes by humans as a psychophysiological problem that involves a relationship between a person and a physical stimulus [11]. When you perceive a real-world object, you make an inductive inference and associate this perception with some general concepts or clues that you have derived from your past experience. Human recognition, in reality, is a question of estimating the relative odds that the input data can be associated with a class from a set of known classes, which depend on our past experiences and clues for recognition. Intelligent classification is intellectually hard work and may seem rather arbitrary. That is how our minds work [6]. Martin and Odell have observed in object-oriented analysis and design that, "In fact, an object can be categorized in more than one way. For example, in Figure 7-1 one person may regard the object Betty as a Woman. Her boss regards her as an Employee. The person who mows her lawn classifies her as a Pet Owner. The local animal control agency licenses her as a Pet Owner. The credit bureau reports that Betty is an instance of the object types called Good Credit Risk—and so on." [6, p. 77]

The problem of classification may be regarded as one of discriminating things, not between the individual objects but between classes, via the search for features or invariant attributes or behaviors among members of a class. Classification can be defined as the categorization of input data (things) into identifiable classes via the extraction of significant features of attributes of the data from a background of irrelevant detail. Another issue in relationships among classes is studied in Chapter 8.
7.3 APPROACHES FOR IDENTIFYING CLASSES

In the following sections, we look at four alternative approaches for identifying classes: the noun phrase approach; the common class patterns approach; the use-case driven, sequence/collaboration modeling approach; and the Classes, Responsibilities, and Collaborators (CRC) approach.

The first two approaches have been included to increase your understanding of the subject; the unified approach uses the use-case driven approach for identifying classes and understanding the behavior of objects. However, you always can combine these approaches to identify classes for a given problem.

Another approach that can be used for identifying classes is Classes, Responsibilities, and Collaborators (CRC) developed by Cunningham, Wilkerson, and Beck. Classes, Responsibilities, and Collaborators, more technique than method, is used for identifying classes responsibilities and therefore their attributes and methods.

7.4 NOUN PHRASE APPROACH

The noun phrase approach was proposed by Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener [12]. In this method, you read through the requirements or use cases looking for noun phrases. Nouns in the textual description are considered to be classes and verbs to be methods of the classes (identifying methods will be covered in Chapter 8). All plurals are changed to singular, the nouns are listed, and the list divided into three categories (see Figure 7–2): relevant classes, fuzzy classes (the “fuzzy area,” classes we are not sure about), and irrelevant classes.

It is safe to scrap the irrelevant classes, which either have no purpose or will be unnecessary. Candidate classes then are selected from the other two categories. Keep in mind that identifying classes and developing a UML class diagram just like other activities is an iterative process. Depending on whether such object modeling is for the analysis or design phase of development, some classes may need to be added or removed from the model and, remember, flexibility is a virtue. You must be able to formulate a statement of purpose for each candidate class; if not, simply eliminate it.

7.4.1 Identifying Tentative Classes

The following are guidelines for selecting classes in an application:

- Look for nouns and noun phrases in the use cases.
- Some classes are implicit or taken from general knowledge.
- All classes must make sense in the application domain; avoid computer implementation classes—defer them to the design stage.
- Carefully choose and define class names.

As explained before, finding classes is not easy. The more practice you have, the better you get at identifying classes. Finding classes is an incremental and iterative process. Booch [2, p. 149] explains this point elegantly: “Intelligent classification is intellectually hard work, and it best comes about through an incremental and iterative process. This incremental and iterative nature is evident in the development of such diverse software technologies as graphical user interfaces,
FIGURE 7-2
Using the noun phrase strategy, candidate classes can be divided into three categories: Relevant Classes, Fuzzy Area or Fuzzy Classes (those classes that we are not sure about), and Irrelevant Classes.

database standards, and even fourth-generation languages.” As Shaw observed [9, p. 143], in software engineering,

the development of individual abstractions often follows a common pattern. First the problems are solved ad hoc. As experience accumulates, some solutions turn out to work better than others, and a sort of folklore is passed informally from person to person. Eventually, the useful solutions are understood more systematically, and they are codified and analyzed. This enables the development of models that support automatic implementation and theories that allow the generalization of the solution. This in turn enables a more sophisticated level of practice and allows us to tackle harder problems—which we often approach ad hoc, starting the cycle over again.

7.4.2 Selecting Classes from the Relevant and Fuzzy Categories
The following guidelines help in selecting candidate classes from the relevant and fuzzy categories of classes in the problem domain.

- **Redundant classes.** Do not keep two classes that express the same information. If more than one word is being used to describe the same idea, select the one that is the most meaningful in the context of the system. This is part of building a common vocabulary for the system as a whole [12]. Choose your vocabulary carefully; use the word that is being used by the user of the system.

- **Adjectives classes.** Wirfs-Brock, Wilkerson, and Wiener warn us about adjectives: “Be wary of the use of adjectives. Adjectives can be used in many ways. An adjective can suggest a different kind of object, different use of the same object, or it could be utterly irrelevant. Does the object represented by the noun behave differently when the adjective is applied to it? If the use of the adjective signals that the behavior of the object is different, then make a new class” [12, p. 38]. For example, Adult Members behave differently than Youth Members, so the two should be classified as different classes.

- **Attribute classes.** Tentative objects that are used only as values should be defined or restated as attributes and not as a class. For example, Client Status and Demographic of Client are not classes but attributes of the Client class.

- **Irrelevant classes.** Each class must have a purpose and every class should be clearly defined and necessary. You must formulate a statement of purpose for each candidate class. If you cannot come up with a statement of purpose, simply eliminate the candidate class.
FIGURE 7-3
The process of eliminating the redundant classes and refining the remaining classes is not sequential. You can move back and forth among these steps as often as you like.

Remember that this is an incremental process. Some classes will be missing, others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably are missing more classes than you will eliminate. Although some classes ultimately may become superclasses, at this stage simply identify them as individual, specific classes. Your design will go through many stages on its way to completion, and you will have adequate opportunity to revise it [12].

Like any other activity of software development, the process of identifying relevant classes and eliminating irrelevant classes is an incremental process. Each iteration often uncovers some classes that have been overlooked. The repetition of the entire process, combined with what you already have learned and the reworking of your candidate classes will enable you to gain a better understanding of the system and the classes that make up your application. Classification is the essence of good object-oriented analysis and design. You must continue this refining cycle through the development process until you are satisfied with the results. Remember that this process (of eliminating redundant classes, classes containing adjectives, possible attributes, and irrelevant classes) is not sequential. You can move back and forth among these steps as often as you like (see Figure 7-3).

7.4.3 The ViaNet Bank ATM System: Identifying Classes by Using Noun Phrase Approach
To better understand the noun phrase method, we will go through a case and apply the noun phrase strategy for identifying the classes. We must start by reading the use cases and applying the principles discussed in this chapter for identifying classes (see Chapter 6 for the description and use cases of the bank system).

7.4.4 Initial List of Noun Phrases: Candidate Classes
The initial study of the use cases of the bank system produces the following noun phrases (candidate classes—maybe).

Account
Account Balance
Amount
Approval Process
ATM Card
ATM Machine
Bank
Bank Client
Card
Cash
Check
Checking
Checking Account
Client
Client’s Account
Currency
Dollar
Envelope
Four Digits
Fund
Invalid PIN
Message
Money
Password
PIN
PIN Code
Record
Savings
Savings Account
Step
System
Transaction
Transaction History

It is safe to eliminate the irrelevant classes. The candidate classes must be selected from relevant and fuzzy classes. The following irrelevant classes can be eliminated because they do not belong to the problem statement: Envelope, Four Digits, and Step. Strikeouts indicate eliminated classes.

Account
Account Balance
Amount
Approval Process
ATM Card
ATM Machine  
Bank  
BankClient  
Card  
Cash  
Check  
Checking  
Checking Account  
Client  
Client's Account  
Currency  
Dollar  
Envelope  
Four Digits  
Fund  
Invalid PIN  
Message  
Money  
Password  
PIN  
PIN Code  
Record  
Savings  
Savings Account  
Step  
System  
Transaction  
Transaction History  

7.4.5 Reviewing the Redundant Classes and Building a Common Vocabulary  

We need to review the candidate list to see which classes are redundant. If different words are being used to describe the same idea, we must select the one that is the most meaningful in the context of the system and eliminate the others.  

The following are the different class names that are being used to refer to the same concept:

Client, BankClient = BankClient (the term chosen)  
Account, Client's Account = Account  
PIN, PIN Code = PIN  
Checking, Checking Account = Checking Account
Savings, Savings Account = Savings Account
Fund, Money = Fund
ATM Card, Card = ATM Card

Here is the revised list of candidate classes:
Account
Account Balance
Amount
Approval Process
ATM Card
Bank
BankClient
Card
Cash
Check
Chequeing
Checking Account
Client
Client's Account
Currency
Dollar
Envelope
Four-Digit
Fund
Invalid PIN
Message
Money
Password
PIN
PIN-Code
Record
Savings
Savings Account
Step
System
Transaction
Transaction History

7.4.6 Reviewing the Classes Containing Adjectives
We again review the remaining list, now with an eye on classes with adjectives. The main question is this: Does the object represented by the noun behave differently
when the adjective is applied to it? If an adjective suggests a different kind of class or the class represented by the noun behaves differently when the adjective is applied to it, then we need to make a new class. However, if it is a different use of the same object or the class is irrelevant, we must eliminate it.

In this example, we have no classes containing adjectives that we can eliminate.

### 7.4.7 Reviewing the Possible Attributes

The next review focuses on identifying the noun phrases that are attributes, not classes. The noun phrases used only as values should be restated as attributes. This process also will help us identify the attributes of the classes in the system.

- **Amount**: A value, not a class.
- **Account Balance**: An attribute of the Account class.
- **Invalid PIN**: It is only a value, not a class.
- **Password**: An attribute, possibly of the BankClient class.
- **Transaction History**: An attribute, possibly of the Transaction class.
- **PIN**: An attribute, possibly of the BankClient class.

Here is the revised list of candidate classes. Notice that the eliminated classes are strikeouts (they have a line through them).

```
Account
Account-Balance
Amount
Approval Process
ATM Card
Bank
BankClient
Card
Cash
Check
Checking
Checking Account
Client
Client's Account
Currency
Dollar
Envelope
Four-Digits
Fund
Invalid-PIN
Message
Money
```
7.4.8 Reviewing the Class Purpose

Identifying the classes that play a role in achieving system goals and requirements is a major activity of object-oriented analysis. Each class must have a purpose. Every class should be clearly defined and necessary in the context of achieving the system’s goals. If you cannot formulate a statement of purpose for a class, simply eliminate it. The classes that add no purpose to the system have been deleted from the list. The candidate classes are these:

ATM Machine class: Provides an interface to the ViaNet bank.
ATMCARD class: Provides a client with a key to an account.
BankClient class: A client is an individual that has a checking account and, possibly, a savings account.
Bank class: Bank clients belong to the Bank. It is a repository of accounts and processes the accounts’ transactions.
Account class: An Account class is a formal (or abstract) class, it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount.
CheckingAccount class: It models a client’s checking account and provides more specialized withdrawal service.
SavingsAccount class: It models a client’s savings account.
Transaction class: Keeps track of transaction, time, date, type, amount, and balance.

No doubt, some classes are missing from the list and others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably will miss more classes than you will eliminate. After all, this is an incremental process; as you learn more about the problem, your design will go through many stages on its way to completion. Remember, there is no such thing as the “right” set of classes. However, the process of identifying classes can improve gradually through this incremental process.

The major problem with the noun phrase approach is that it depends on the completeness and correctness of the available document, which is rare in real life. On the other hand, large volumes of text on system documentation might lead to too many
candidate classes. Even so, the noun phrase exercise can be very educational and useful if combined with other approaches, especially with use cases as we did here.

7.5 COMMON CLASS PATTERNS APPROACH

The second method for identifying classes is using common class patterns, which is based on a knowledge base of the common classes that have been proposed by various researchers, such as Shlaer and Mellor [10], Ross [8], and Coad and Yourdon [3]. They have compiled and listed the following patterns for finding the candidate class and object:

- **Name. Concept class**
  
  *Context.* A concept is a particular idea or understanding that we have of our world. The concept class encompasses principles that are not tangible but used to organize or keep track of business activities or communications. Martin and Odell [6, p. 236] describe concepts elegantly, “Privately held ideas or notions are called conceptions. When an understanding is shared by another, it becomes a concept. To communicate with others, we must share our individually held conceptions and arrive at agreed concepts.” Furthermore, Martin and Odell explain that, without concepts, mental life would be total chaos since every item we encountered would be different.
  
  *Example.* Performance is an example of concept class object.

- **Name. Events class**
  
  *Context.* Events classes are points in time that must be recorded. Things happen, usually to something else at a given date and time or as a step in an ordered sequence. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why.
  
  *Example.* Landing, interrupt, request, and order are possible events.

- **Name. Organization class**
  
  *Context.* An organization class is a collection of people, resources, facilities, or groups to which the users belong; their capabilities have a defined mission, whose existence is largely independent of the individuals.
  
  *Example.* An accounting department might be considered a potential class.

- **Name. People class** (also known as person, roles, and roles played class)
  
  *Context.* The people class represents the different roles users play in interacting with the application. People carry out some function. What role does a person play in the system? Coad and Yourdon [3] explain that a class which is represented by a person can be divided into two types: those representing users of the system, such as an operator or clerk who interacts with the system; and those representing people who do not use the system but about whom information is kept by the system.
  
  *Example.* Employee, client, teacher, and manager are examples of people.

- **Name. Places class**
  
  *Context.* Places are physical locations that the system must keep information about.
Example. Buildings, stores, sites, and offices are examples of places.

- Name. **Tangible things and devices class**
  
  Context. This class includes physical objects or groups of objects that are tangible and devices with which the application interacts.
  
  Example. Cars are an example of tangible things, and pressure sensors are an example of devices.

### 7.5.1 The ViaNet Bank ATM System: Identifying Classes by Using Common Class Patterns

To better understand the common class patterns approach, we once again will try to identify classes in the bank system by applying common class patterns. The common class patterns are concepts, events, organization, people, places, and tangible things and devices.

**Events** classes are points in time that must be recorded. Associated with things remembered are attributes (after all, the things to remember are objects) such as who, what, when, where, how, or why. The bank system events classes follow.

- **Account class:** An Account class is a *formal* (or *abstract*) class; it defines the common behaviors that can be inherited by more specific classes such as CheckingAccount and SavingsAccount.
- **CheckingAccount class:** It models a client's checking account and provides more specialized withdrawal service.
- **SavingsAccount class:** It models a client's savings account.
- **Transaction class:** It keeps track of transaction, time, date, type, amount, and balance.

**Organization** classes specify collections of people, resources, facilities, or groups to which the users belong, and their capabilities have a defined mission, whose existence is largely independent of individuals. The bank system's organization class follows.

- **Bank class:** Bank clients belong to the Bank. It is a repository of accounts and processes the accounts' transactions.

**People and person** classes answer this question: What role does a person play in the system? Coad and Yourdon [3] explain that a class being represented by a person can be divided into two types: those representing the users of the system, such as an operator or a clerk who interacts with systems, and those people who do not use the system but about whom information is kept by the system. The following is the bank system people and person class.

- **BankClient class:** A client is an individual that has a checking account and, possibly, a savings account.

**Place** classes represent physical locations, buildings, stores, sites, or offices about which the system needs to keep track. Place classes are not applicable to this bank system.
The tangible things and devices classes represent physical objects or groups of objects that are tangible and devices with which the application interacts. In the banking system, tangible and device classes include these items.

ATM Machine class: It allows access to all accounts held by a bank client.

7.6 USE-CASE DRIVEN APPROACH: IDENTIFYING CLASSES AND THEIR BEHAVIORS THROUGH SEQUENCE/COLLABORATION MODELING

The use-case driven approach is the third approach that we examine in this chapter and the one that is recommended. From the previous chapter, we learned that use cases are employed to model the scenarios in the system and specify what external actors interact with the scenarios. The scenarios are described in text or through a sequence of steps. Use-case modeling is considered a problem-driven approach to object-oriented analysis, in that the designer first considers the problem at hand and not the relationship between objects, as in a data-driven approach.

Modeling with use cases is a recommended aid in finding the objects of a system and is the technique used by the unified approach. Once the system has been described in terms of its scenarios, the modeler can examine the textual description or steps of each scenario to determine what objects are needed for the scenario to occur. However, this is not a magical process in which you start with use cases, develop a sequence diagram, and voilà, classes appear before your eyes. The process of creating sequence or collaboration diagrams is a systematic way to think about how a use case (scenario) can take place; and by doing so, it forces you to think about objects involved in your application.

When building a new system, designers model the scenarios of the way the system of business should work. When redesigning an existing system, many modelers choose to first model the scenarios of the current system, and then model the scenarios of the way the system should work. Developing scenarios also requires us to think about class methods, which will be studied in Chapter 8.

7.6.1 Implementation of Scenarios

The UML specification recommends that at least one scenario be prepared for each significantly different use-case instance. Each scenario shows a different sequence of interaction between actors and the system, with all decisions definite. In essence, this process helps us to understand the behavior of the system's objects.

When you have arrived at the lowest use-case level, you may create a child sequence diagram or accompanying collaboration diagram for the use case. With the sequence and collaboration diagrams, you can model the implementation of the scenario [7].

Like use-case diagrams, sequence diagrams are used to model scenarios in the systems. Whereas use cases and the steps or textual descriptions that define them offer a high-level view of a system, the sequence diagram enables you to model a more specific analysis and also assists in the design of the system by modeling the interactions between objects in the system.
As explained in Chapter 5, in a sequence diagram, the objects involved are drawn on the diagram as a vertical dashed line, with the name of the objects at the top. Horizontal lines corresponding to the events that occur between objects are drawn between the vertical object lines. The event lines are drawn in sequential order, from the top of the diagram to the bottom. They do not necessarily correspond to the steps defined for a use-case scenario.

7.6.2 The ViaNet Bank ATM System: Decomposing a Use-Case Scenario with a Sequence Diagram: Object Behavior Analysis

A sequence diagram represents the sequence and interactions of a given use case or scenario. Sequence diagrams are among the most popular UML diagrams and, if used with an object model or class diagram, can capture most of the information about a system [5]. Most object-to-object interactions and operations are considered events, and events include signals, inputs, decisions, interrupts, transitions, and actions to or from users or external devices. An event also is considered to be any action by an object that sends information. The event line represents a message sent from one object to another, in which the “from” object is requesting an operation be performed by the “to” object. The “to” object performs the operation using a method that its class contains. Developing sequence or collaboration diagrams requires us to think about objects that generate these events and therefore will help us in identifying classes.

To identify objects of a system, we further analyze the lowest level use cases with a sequence and collaboration diagram pair (actually, most CASE tools such as SA/Object allow you to create only one, either a sequence or a collaboration diagram, and the system generates the other one). Sequence and collaboration diagrams represent the order in which things occur and how the objects in the system send messages to one another. These diagrams provide a macro-level analysis of the dynamics of a system. Once you start creating these diagrams, you may find that objects may need to be added to satisfy the particular sequence of events for the given use case.

You can draw sequence diagrams to model each scenario that exists when a BankClient withdraws, deposits, or needs information on an account. By walking through the steps, you can determine what objects are necessary for those steps to take place. Therefore, the process of creating sequence or collaboration diagrams can assist you in identifying classes or objects of the system. This approach can be combined with noun phrase and class categorization for the best results.

In Chapter 6, we identified the use cases for the bank system. The following are the low level (executable) use cases:

- Deposit Checking
- Deposit Savings
- Invalid PIN
- Withdraw Checking
- Withdraw More from Checking
- Withdraw Savings
- Withdraw Savings Denied
Checking Transaction History
Savings Transaction History

Let us create a sequence/collaboration diagram for the following use cases:

- Invalid PIN use case
- Withdraw Checking use case
- Withdraw More from Checking use case

Sequence/collaboration diagrams are associated with a use case. For example, to model the sequence/collaboration diagrams in SA/Object, you must first select a use case, such as the Invalid PIN use case, then associate a sequence or collaboration child process.

To create a sequence you must think about the classes that probably will be involved in a use-case scenario. Keep in mind that use case refers to a process, not a class. However, a use case can contain many classes, and the same class can occur in many different use cases. Point of caution: you should defer the interfaces classes to the design phase and concentrate on the identifying business classes here.

Consider how we would prepare a sequence diagram for the Invalid PIN use case. Here, we need to think about the sequence of activities that the actor BankClient performs:

- Insert ATM Card.
- Enter PIN number.
- Remove the ATM Card.

Based on these activities, the system should either grant the access right to the account or reject the card. Next, we need to more explicitly define the system. With what are we interacting? We are interacting with an ATMMachine and the BankClient. So, the other objects of this use case are ATMMachine and BankClient.

Now that we have identified the objects involved in the use case, we need to list them in a line along the top of a page and drop dotted lines beneath each object (see Figure 7–4). The client in this case is whoever tries to access an account

**FIGURE 7–4**
The sequence diagram for the Invalid PIN use case.
through the ATM, and may or may not have an account. The BankClient on the other hand has an account.

The dotted lines are the lifelines discussed in Chapter 5. The line on the right represents an actor, in this case the BankClient, or an event that is outside the system boundary. Recall from Chapter 5 that an event arrow connect objects. In effect, the event arrow suggests that a message is moving between those two objects. An example of an event message is the request for a PIN. An event line can pass over an object without stopping at that object. Each event must have a descriptive name. In some cases, several objects are active simultaneously, even if they are only waiting for another object to return information to them. In other cases, an object becomes active when it receives a message and then becomes inactive as soon as it responds [5]. Similarly, we can develop sequence diagrams for other use cases (as in Figures 7–5 and 7–7). Collaboration diagrams are just another view of the sequence diagrams and therefore can be created automatically; most UML modeling tools automatically create them (see Figures 7–6 and 7–8).

The following classes have been identified by modeling the UML sequence/collaboration diagrams: Bank, BankClient, ATMMachine, Account, Checking Account, and Savings Account.

Similarly other classes can be identified by developing the remaining sequence/collaboration diagrams. Developing the other sequence/collaboration diagrams has been left as an exercise; see problem 1–3.

**FIGURE 7–5**
Sequence diagram for the Withdraw Checking use case.
FIGURE 7-6
The collaboration diagram for the Withdraw Checking use case.

FIGURE 7-7
The sequence diagram for the Withdraw More from Checking use case.
7.7 CLASSES, RESPONSIBILITIES, AND COLLABORATORS

Classes, responsibilities, and collaborators (CRC), developed by Cunningham, Wilkerson, and Beck, was first presented as a way of teaching the basic concepts of object-oriented development [12]. Classes, Responsibilities, and Collaborators is a technique used for identifying classes' responsibilities and therefore their attributes and methods. Furthermore, Classes, Responsibilities, and Collaborators can help us identify classes. Classes, Responsibilities, and Collaborators is more a teaching technique than a method for identifying classes. Classes, Responsibilities, and Collaborators is based on the idea that an object either can accomplish a certain responsibility itself or it may require the assistance of other objects. If it requires the assistance of other objects, it must collaborate with those objects to fulfill its responsibility [13]. By identifying an object's responsibilities and collaborators (cooperative objects with which it works) you can identify its attributes and methods.

Classes, Responsibilities, and Collaborators cards are 4" x 6" index cards. All the information for an object is written on a card, which is cheap, portable, readily available, and familiar. Figure 7–9 shows an idealized card. The class name should appear in the upper left-hand corner, a bulleted list of responsibilities should appear under it in the left two thirds of the card, and the list of collaborators should appear in the right third. However, rather than simply tracing the details of a collaboration in the form of message sending, Classes, Responsibilities, and Collaborators cards place the designer's focus on the motivation for collaboration by representing (potentially) many messages as phrases of English text.
Classes, Responsibilities, and Collaborators stresses the importance of creating objects, not to meet mythical future needs, but only under the demands of the moment. This ensures that a design contains only as much information as the designer has directly experienced and avoids premature complexity. Working in teams helps here, because a concerned designer can influence team members by suggesting scenarios aimed specifically at suspected weaknesses or omissions.

7.7.1 Classes, Responsibilities, and Collaborators Process

The Classes, Responsibilities, and Collaborators process consists of three steps (see Figure 7–10) [1]:

1. Identify classes' responsibilities (and identify classes).
2. Assign responsibilities.
3. Identify collaborators.

Classes are identified and grouped by common attributes, which also provides candidates for superclasses. The class names then are written onto Classes, Responsibilities, and Collaborators cards. The card also notes sub- and superclasses to show the class structure. The application's requirements then are examined for actions and information associated with each class to find the responsibilities of each class.

Next, the responsibilities are distributed; they should be as general as possible and placed as high as possible in the inheritance hierarchy.

The idea in locating collaborators is to identify how classes interact. Classes (cards) that have a close collaboration are grouped together physically.

FIGURE 7–10
The Classes, Responsibilities, and Collaborators process.
7.7.2 The ViaNet Bank ATM System: Identifying Classes by Using Classes, Responsibilities, and Collaborators

We already identified the initial classes of the bank system. The objective of this example is to identify objects' responsibilities such as attributes and methods in that system.

Account and Transaction provide the banking model. Note that Transaction assumes an active role while money is being dispensed and a passive role thereafter.

The class Account is responsible mostly to the BankClient class and it collaborates with several objects to fulfill its responsibilities. Among the responsibilities of the Account class to the BankClient class is to keep track of the BankClient balance, account number, and other data that need to be remembered. These are the attributes of the Account class. Furthermore, the Account class provides certain services or methods, such as means for BankClient to deposit or withdraw an amount and display the account's Balance (see Figure 7-11).

Classes, Responsibilities, and Collaborators encourages team members to pick up the card and assume a role while "executing" a scenario. It is not unusual to see a designer with a card in each hand, waving them about, making a strong identification with the objects while describing their collaboration. Ward Cunningham writes:

Classes, Responsibilities, and Collaborators cards work by taking people through programming episodes together. As cards are written for familiar objects, all participants pick up the same context and ready themselves for decision making. Then, by waving cards and pointing fingers and yelling statements like, "no, this guy should do that," decisions are made. Finally, the group starts to relax as consensus has been reached and the issue becomes simply finding the right words to record a decision as a responsibility on a card.

In similar fashion other cards for the classes that have been identified earlier in this chapter must be created, with the list of their responsibilities and their collaborators. As you can see from Figure 7-10, this process is iterative.

Start with few cards (classes) then proceed to play "what if." If the situation calls for a responsibility not already covered by one of the objects, either add the responsibility to an object or create a new object to address that responsibility. If one of the objects becomes too cluttered during this process, copy the information on

FIGURE 7-11
Classes, Responsibilities, and Collaborators for the Account object.

<table>
<thead>
<tr>
<th>Account</th>
<th>Checking Account</th>
</tr>
</thead>
<tbody>
<tr>
<td>balance</td>
<td>(subclass)</td>
</tr>
<tr>
<td>number</td>
<td>Savings Account</td>
</tr>
<tr>
<td>deposit</td>
<td>(subclass)</td>
</tr>
<tr>
<td>withdraw</td>
<td>Transaction</td>
</tr>
<tr>
<td>getBalance</td>
<td></td>
</tr>
</tbody>
</table>

1This section is adapted from "Laboratory for Teaching Object-Oriented Thinking" by Kent Beck and Ward Cunningham, with permission of the Association for Computing Machinery.
Box 7.1

Real-World Issues on the Agenda
CRC: HOW DO TEAMS SHAPE OBJECTS? HOW DO OBJECTS SHAPE TEAMS?

Ward Cunningham

CRC stands for Classes, Responsibilities, and Collaborators. My colleague, Kent Beck, and I would struggle to coax a program into existence sitting side-by-side, sharing the controls of a then new Smalltalk workstation. We found repeatedly that our problems would yield when we could articulate the responsibilities of the objects we had made and the collaborations they would use to meet those responsibilities. I had many months' experience with such grappling before I ever tried to duplicate the sensation away from a computer. The first ever CRC Cards were written at 4:00 AM one morning on my dining room table. The first ever team development of CRC Cards took place around 9:00 AM that same day as Brian Wilkerson and I grappled with responsibilities at a cafeteria table 1000 feet from our machines. As with Kent many times before, here I was coaxing a program into existence. I'd like to leave CRC for the moment and instead focus on what I mean by coaxing a program into existence.

A program is decision made. Humans make those decisions and write them into programs. When I compare the research programming I did early on to the production programming I’ve done recently, I find that the activities differ only in the quantity of decisions made on a given day. In both cases decisions come in spurts. A period of investigation precedes a spurt of decisions. Likewise, a period of rather mechanical activity following through with consequences follows a decision spurt. The feeling one gets is of lifting a problem up into the mind, struggling with it until decisions come, then putting those decisions down in place of the problem. Only then can one relax. This cycle repeats. It’s what I call grappling. I feel it when I program, with cards or at a machine.

I now call these grapple cycles programming episodes. Episodic programming is most noticeable when decisions require judgment. Mechanical decisions, like looking something up in a book, don’t feel particularly episodic. But, then, they aren’t actually decisions. Designing framework software on the other hand does require judgment and progresses in distinct episodes.

CRC Cards work by taking people through programming episodes together. As cards are written for familiar objects, all participants pick up the same context and ready themselves for decision making. Then, by waving cards and pointing fingers and yelling statements like, “no, this guy should do that,” decisions are made. Finally, the group starts to relax as consensus has been reached and the issue becomes simply finding the right words to record a decision as a responsibility on a card.

I’ve had people tell me about their struggles to make CRC Cards work. Often many none-too-fruitful sessions preceded an almost spiritual experience when the right people are finally assembled and they “come together” to break a logjam of decision.

Such reports are, of course, satisfying. Even more satisfying is my more recent experience in front of the Smalltalk browsers. Having left research, I found myself swimming in a river of decisions. As I pointed out earlier, production programming re-

its card to a new card, searching for more concise ways of saying what the object does. If it is not possible to shrink the information further and the object is still too complex, create a new object to assume some of the responsibilities.

7.8 Naming Classes

Naming a class is an important activity. Here are guidelines for naming classes:

- The class name should be singular. The class should describe a single object, so it should be the singular form of noun (it may be an adjective and a noun, such as YouthMember).
quires many times more decisions per day than research. Production decisions are rarely as profound as those in research, but they are of equal or greater consequence and demand as much or more judgment. Our production team “came together” when we learned to work through episodes promptly and in synchronization so that our collective experience came to bear on all of our decision making.

We regularly worked two to a machine. This worked best when we had enough framework in place that our Smalltalk code read like specification. An episode would begin by poking around with “senders,” “implementers,” a few “inspectors” and an occasional “ctrl-c.” We were lifting the problem into our collective consciousness. One would type, the other watch, then trade off. As we approached decisions, our attention would turn to each other. The communication would become complex as human communication often does. I won’t attempt to analyze it except to say that it includes a lot of hand waving, body motion and statements like, “hey, that guy shouldn’t do that.” Finally, decisions would be made and the consequences followed through, not by writing on cards, but by adjusting the specification-like code. And, as with cards, we would search for the right words to represent the decisions as they had come to us.

What I’ve just described, I’d felt in research. And, I thought of it as just a happy compatibility. In production, our machine sharing collaborations were more intentional, based on our need to work quickly and correctly. It was a practice that grew slowly in our core group of four implementers, individuals of only average compatibility. The practice spread by firsthand experience. First, others learned that I preferred collaborative episodes as a work style. Then, they chose to work that way, too. Ultimately, any possible pairing was likely in the office and a group of three would form when the problems required that much talent.

I’d like to point out again the unusually high capacity human-to-human communication path that is opened by two people sharing a machine. As the developers work through whole programming episodes together, the machine presents a broad range of situations to be dealt with, all, of course, relevant to the work at hand. In order to stay synchronized each programmer must at least follow the problem-solving strategies and techniques of the other. Things that worked were obvious. These spread quickly as did our general understanding of what worked well and what didn’t in the program as a whole.

I now recognize the organization we built to be a High-Performance Team. Such teams are recognized by their ability to play to members strengths while covering for each other’s weaknesses. We were able to work at maximum productivity continuously and indefinitely. When an occasional crisis did occur, we simply pulled the most relevant people together and worked through the problem as another episode. We knew we were working on the most important thing in the most productive way. No further urgency was necessary.

In summary, Smalltalk’s specification-like coding and incremental development environment permits a unique human-human-machine dialogue. CRC Cards allow larger groups to feel this dialogue which is based on repeated episodes of decision making. Finally, members of somewhat larger production development teams can exploit the human-human-machine dialogue on a pair-wise basis. The complex communication that then takes place will support a High-Performance organization with many benefits, a few of which I have mentioned.

- One general rule for naming classes is that you should use names with which the users or clients are comfortable. Choose the class name from standard vocabulary for the subject matter. Select the name with which the client is comfortable rather than semantically accurate terminology [3].
- The name of a class should reflect its intrinsic nature. Stick with the general terminology of the subject matter.
- Use readable names. Capitalize class names. By convention, the class name must begin with an upper-case letter. For compound words, capitalize the first letter of each word; for example, LoanWindow. While these conventions are not mandatory, they make the code consistent and easy to read. Do not add prefix
and suffix codes, they are a bother to read and troublesome when they need to be changed [3].

7.9 SUMMARY

Object analysis is a process by which we can identify the classes that play a role in achieving the system goals and requirements. The problem of classification may be regarded as one of discriminating things, not between the individual objects, but between classes via the search for features or invariant attributes (or behaviors) among members of a class.

Finding classes is one of the hardest activities in object-oriented analysis. Unfortunately classes are rarely just there for the picking. There is no such thing as the perfect class structure or the right set of objects. Nevertheless, several approaches—such as the use-case driven approach, the noun phrase, and the knowledge base of common class patterns, and Classes, Responsibilities, and Collaborators—can offer us guidelines and general rules for identifying the classes in the given problem domain. Furthermore, identifying classes is an iterative process, and as you gain more experience, you get better at identifying classes.

In this chapter, we studied four approaches for identifying classes: the noun phrase, common class patterns, use-case driven, and Classes, Responsibilities, and Collaborators. The process of identifying classes can improve gradually through the incremental process. Some classes will be missing in the first few cycles of identification, and others will be eliminated or refined later. Unless you are starting with a lot of domain knowledge, you probably will miss more classes than you will eliminate. Your design will go through many stages on its way to completion as you learn more about the problem.

To identify classes using the noun phrase approach, read through use cases, looking for noun phrases. Consider nouns in the textual description to be classes and verbs to be methods of the classes.

The second method for identifying classes is the common class patterns approach based on the knowledge base of the common classes proposed by various researchers. These researchers compiled and listed several categories for finding the candidate classes and objects.

The third method we studied was use-case driven. To identify objects of a system and their behaviors, the lowest level of executable use cases is further analyzed with a sequence and collaboration diagram pair. By walking through the steps, you can determine what objects are necessary for the steps to take place. Finally, we looked at the Classes, Responsibilities, and Collaborators, which is a useful tool for learning about class responsibilities and identifying classes. These approaches can be mixed for identifying the classes of a given problem.

Naming a class is an important activity, too. The class should describe a single object, so it should be a singular noun or an adjective and a noun. A general rule for naming classes is that you use names with which the users or clients are comfortable. Choose the class names from standard vocabulary for the subject matter.
KEY TERMS

Adjective class (p. 155)
Attribute class (p. 155)
Classes, Responsibilities, and Collaborators (CRC) (p. 169)
Classification (p. 152)
Collaborator (p. 169)
Common class patterns (p. 162)
Concepts class (p. 162)
Events class (p. 162)
Irrelevant class (p. 155)
Organization class (p. 162)
Places class (p. 162)
People (person) class (p. 162)
Redundant class (p. 155)
Tangible things and device class (p. 163)

REVIEW QUESTIONS

1. Where do objects come from?
2. Describe the noun phrase strategy for identifying tentative classes in a problem domain.
3. Describe relevant, fuzzy, and irrelevant classes.
4. How would you select candidate classes for the list of relevant and fuzzy classes?
5. What criteria would you use to eliminate a class?
6. What is the common class patterns strategy?
7. What clues would you use to identify classes?
8. How would you name classes?
9. What are the concepts classes?
10. What are the events classes?
11. What are the organization classes?
12. What are the other system classes?
13. What are the people (person) classes?
14. Explain the places class source.
15. What are the tangible things and devices classes?
16. Why is identifying classes an incremental process?
17. Why is developing a sequence/collaboration diagram a useful activity in identifying classes?
18. Why is Classes, Responsibilities, and Collaborators useful?

PROBLEMS

1. Develop sequence/collaboration diagrams for the Deposit Checking use case of the bank system.
2. Develop sequence/collaboration diagrams for the Deposit Savings use case of the bank system.
3. Develop sequence/collaboration diagrams for the Withdraw Savings Denied use case of the bank system.
4. Identify classes in the Grandma’s Soup Unlimited problem (see Chapter 6) by using one
(or a combination) of approaches you have learned in this chapter. Also, write a reason(s)
for the approach that you have selected.

REFERENCES

1. Anderson, Michael; and Bergstrand, John. “Formalizing Use Cases with Message Se-
quence Charts.” Master thesis, Department of Communication Systems, Lund Institute
2. Booch, Grady. *Object-Oriented Analysis and Design with Applications.* Redwood, CA:
3. Coad, Peter; and Yourdon, Edward. *Object-Oriented Analysis*, 2d ed. Englewood Cliffs,
4. Gabriel, R.; White, J.; and Bobrow, D. “CLOS Integrating Object-Oriented and Func-
tional Programming.” *Communications of the ACM* 34, no. 9.
5. Harmon, P.; and Watson, M. *Understanding UML: The Developer’s Guide with a Web-
6. Martin, James; and Odell, James. *Object-Oriented Analysis and Design.* Englewood
9. Shaw, M. “Large Scale Systems Require Higher-Level Abstractions.” *SIGSOFT Engi-
eineering Notes* 14, no. 3 (May 1989), p.143.
10. Shlaer, S.; and Mellor, S. *Object-Oriented System Analysis: Modeling the World in Data.*
11. Tou, J. T.; and Gonzalez, R. C. *Pattern Recognition Principles.* Reading, MA: Addison-
Wesley, 1981.
12. Beck, Kent; and Cunningham, Ward. “A laboratory for teaching Object-Oriented
Thinking,” *Object-Oriented Programming System Languages and Application OOP-
SLA’89*, October 1-6, 1989, New Orleans, LA.
13. Wirfs-Brock, Rebecca; Wilkerson, Brian; and Wiener, Lauren. *Designing Object-Oriented
CHAPTER 8

Identifying Object Relationships, Attributes, and Methods

*Objects contribute to the behavior of the system by collaborating with one another.*

—Grady Booch [2]

Chapter Objectives

You should be able to define and understand
- Analyzing relationships among classes.
- Identifying association.
- Association patterns.
- Identifying super- and subclass hierarchies.
- Identifying aggregation or a-part-of compositions.
- Class responsibilities.
- Identifying attributes and methods by analyzing use cases and other UML diagrams.

8.1 INTRODUCTION

In an object-oriented environment, objects take on an active role in a system. Of course, objects do not exist in isolation but interact with each other. Indeed, these interactions and relationships are the application. All objects stand in relationship to others on whom they rely for services and control. The relationship among objects is based on the assumptions each makes about the other objects, including what operations can be performed and what behavior results [2]. Three types of relationships among objects are

- Association. How are objects associated? This information will guide us in designing classes.
- Super-sub structure (also known as generalization hierarchy). How are objects organized into superclasses and subclasses? This information provides us the direction of inheritance.
• *Aggregation and a-part-of structure.* What is the composition of complex classes? This information guides us in defining mechanisms that properly manage object-within-object [6].

Generally speaking, the relationships among objects are known as *associations.* For example, a customer places an *order* for soup. The *order* is the association between the customer and soup objects. The hierarchical or super-sub relation allows the sharing of properties or inheritance. *A-part-of structure* is a familiar means of organizing components of a bigger object. For example, walls, windows, doors, and the like are part of a bigger object: a building.

In this chapter, we look at guidelines for identifying association, super-sub, and a-part-of relationships in the problem domain. We then proceed to identify attributes and methods. To do this we must first determine the responsibilities of the system. We saw that the system’s responsibilities can be identified by analyzing use cases and their sequence and collaboration diagrams. Once you have identified the system’s responsibilities and what information the system needs to remember, you can assign each responsibility to the class to which it logically belongs. This also aids in determining the purpose and role each class plays in the application.

### 8.2 ASSOCIATIONS

*Association* represents a physical or conceptual connection between two or more objects. For example, if an object has the responsibility for telling another object that a credit card number is valid or invalid, the two classes have an association. In Chapter 5, we learned that the binary associations are shown as lines connecting two class symbols. Ternary and higher-order associations are shown as diamonds connecting to a class symbol by lines, and the association name is written above or below the line. The association name can be omitted if the relationship is obvious. In some cases, you will want to provide names for the roles played by the individual classes making up the relationship. The role name on the side closest to each class describes the role that class plays relative to the class at the other end of the line, and vice versa [4] (see Figure 8–1).

**FIGURE 8–1**
Basic association. See Chapter 5 for a detailed discussion of association.
8.2.1 Identifying Associations

Identifying associations begins by analyzing the interactions between classes. After all, any dependency relationship between two or more classes is an association [7]. You must examine the responsibilities to determine dependencies. In other words, if an object is responsible for a specific task (behavior) and lacks all the necessary knowledge needed to perform the task, then the object must delegate the task to another object that possesses such knowledge. Wirfs-Brock, Wilkerson, and Wiener [8] provide the following questions that can help us to identify associations:

- Is the class capable of fulfilling the required task by itself?
- If not, what does it need?
- From what other class can it acquire what it needs?

Answering these questions helps us identify association. The approach you should take to identify association is flexibility. First, extract all candidates' associations from the problem statement and get them down on paper. You can refine them later. Notice that a-part-of structures (aggregation) and associations are very similar. So, how do you distinguish one from the other? It depends on the problem domain; after all, a-part-of structure is a special case of association. Simply pick the one most natural for the problem domain. If you can represent the problem more easily with association, then select it; otherwise, use a-part-of structure, which is described later in the chapter.

8.2.2 Guidelines for Identifying Association

The following are general guidelines for identifying the tentative associations:

- A dependency between two or more classes may be an association. Association often corresponds to a verb or prepositional phrase, such as part of, next to, works for, or contained in.
- A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

8.2.3 Common Association Patterns

The common association patterns are based on some of the common associations defined by researchers and practitioners: Rumbaugh et al. [7], Coad and Yourdon [3], and others. These include

- Location association—next to, part of, contained in. For example, consider a soup object, cheddar cheese is a-part-of soup. The a-part-of relation is a special type of association, discussed in more detail later in the chapter.
- Communication association—talk to, order to. For example, a customer places an order (communication association) with an operator person (see Figure 8–2).

These association patterns and similar ones can be stored in the repository and added to as more patterns are discovered. However, currently, this capability of the unified approach’s repository is more conceptual than real, but it is my hope that CASE tool vendors in the near future will provide this capability.
8.2.4 Eliminate Unnecessary Associations

- **Implementation association.** Defer implementation-specific associations to the design phase. Implementation associations are concerned with the implementation or design of the class within certain programming or development environments and not relationships among business objects (see Chapter 6 for a definition of business objects).

- **Ternary associations.** Ternary or $n$-ary association is an association among more than two classes (see Chapter 5). Ternary associations complicate the representation. When possible, restate ternary associations as binary associations.

- **Directed actions (or derived) association.** Directed actions (derived) associations can be defined in terms of other associations. Since they are redundant, avoid these types of association. For example, Grandparent of can be defined in terms of the parent of association (see Figure 8-3).

Choose association names carefully. Do not say how or why a situation came about; say what it is. Add role names where appropriate, especially to distinguish multiple associations. These often are discovered by testing access paths to objects.

**Figure 8-3**
Grandparent of Ken can be defined in terms of the parent association.
8.3 SUPER-SUB CLASS RELATIONSHIPS

The other aspect of classification (see Chapter 7) is identification of super-sub relations among classes. For the most part, a class is part of a hierarchy of classes, where the top class is the most general one and from it descend all other, more specialized classes. The super-sub class relationship represents the inheritance relationships between related classes, and the class hierarchy determines the lines of inheritance between classes. Class inheritance is useful for a number of reasons. For example, in some cases, you want to create a number of classes that are similar in all but a few characteristics. In other cases, someone already has developed a class that you can use, but you need to modify that class. Subclasses are more specialized versions of their superclasses. The classes are not ordered this way for convenience’s sake.

Superclass-subclass relationships, also known as generalization hierarchy, allow objects to be built from other objects. Such relationships allow us to explicitly take advantage of the commonality of objects when constructing new classes. The super-sub class hierarchy is a relationship between classes, where one class is the parent class of another (derived) class. Recall from Chapter 2 that the parent class also is known as the base or super class or ancestor. The super-sub class hierarchy is based on inheritance, which is programming by extension as opposed to programming by reinvention [5]. The real advantage of using this technique is that we can build on what we already have and, more important, reuse what we already have. Inheritance allows classes to share and reuse behaviors and attributes. Where the behavior of a class instance is defined in that class’s methods, a class also inherits the behaviors and attributes of all of its superclasses. Now let us take a look at guidelines for identifying classes.

8.3.1 Guidelines for Identifying Super-Sub Relationship, a Generalization

The following are guidelines for identifying super-sub relationships in the application:

- **Top-down.** Look for noun phrases composed of various adjectives in a class name. Often, you can discover additional special cases. Avoid excessive refinement. Specialize only when the subclasses have significant behavior. For example, a phone operator employee can be represented as a cook as well as a clerk or manager because they all have similar behaviors.

- **Bottom-up.** Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class. You may have to alter the definitions a bit; this is acceptable as long as generalization truly applies. However, do not force classes to fit a preconceived generalization structure.

- **Reusability.** Move attributes and behaviors (methods) as high as possible in the hierarchy. At the same time, do not create very specialized classes at the top of the hierarchy. This is easier said than done. The balancing act can be achieved through several iterations. This process ensures that you design objects that can be reused in another application.
Multiple inheritance. Avoid excessive use of multiple inheritance. Multiple inheritance brings with it complications such as how to determine which behavior to get from which class, particularly when several ancestors define the same method. It also is more difficult to understand programs written in a multiple inheritance system. One way of achieving the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as an attribute (see Figure 8–4, aggregation; we will look at this issue in Chapter 9). However, use multiple inheritance when it is appropriate. For example, if the owner of a restaurant prepares the soups, you can utilize multiple inheritance structure to define an OwnerOperator class that inherits its attributes and methods from both the Owner and Operator classes.

8.4 A-PART-OF RELATIONSHIPS—AGGREGATION

A-part-of relationship, also called aggregation, represents the situation where a class consists of several component classes. A class that is composed of other classes does not behave like its parts; actually, it behaves very differently. For example, a car consists of many other classes, one of which is a radio, but a car does not behave like a radio (see Figure 8–5).

Two major properties of a-part-of relationship are transitivity and antisymmetry [7]:

- **Transitivity.** The property where, if A is part of B and B is part of C, then A is part of C. For example, a carburetor is part of an engine and an engine is part of a car; therefore, a carburetor is part of a car. Figure 6–3 shows a-part-of structure.

- **Antisymmetry.** The property of a-part-of relation where, if A is part of B, then B is not part of A. For example, an engine is part of a car, but a car is not part of an engine.

A clear distinction between the part and the whole can help us determine where responsibilities for certain behavior must reside. This is done mainly by asking the following questions [3]:

![Figure 8-4](image-url)
CHAPTER 8: IDENTIFYING OBJECT RELATIONSHIPS, ATTRIBUTES, AND METHODS

FIGURE 8–5
A part-of composition. A carburetor is a part of an engine and an engine and a radio are parts of a car.

- Does the part class belong to a problem domain?
- Is the part class within the system’s responsibilities?
- Does the part class capture more than a single value? (If it captures only a single value, then simply include it as an attribute with the whole class.)
- Does it provide a useful abstraction in dealing with the problem domain?

In Chapter 5, we saw that the UML uses hollow or filled diamonds to represent aggregations. A filled diamond signifies the strong form of aggregation, which is composition. For example, one might represent aggregation such as container and collection as hollow diamonds (see Figures 8–6 and 8–7) and use a solid diamond to represent composition, which is a strong form of aggregation (see Figure 8–5).

8.4.1 A-Part-of Relationship Patterns
To identify a-part-of structures, Coad and Yourdon [3] provide the following guidelines:

- **Assembly.** An assembly is constructed from its parts and an assembly-part situation physically exists; for example, a French onion soup is an assembly of onion, butter, flour, wine, French bread, cheddar cheese, and so on.
- **Container.** A physical whole encompasses but is not constructed from physical parts; for example, a house can be considered as a container for furniture and appliances (see Figure 8–6).

FIGURE 8–6
A house is a container.
8.5 CASE STUDY: RELATIONSHIP ANALYSIS FOR THE VIANET BANK ATM SYSTEM

To better gain experience in object relationship analysis, we use the familiar bank system case and apply the concepts in this chapter for identifying associations, super-sub relationships, and a-part-of relationships for the classes identified in Chapter 7.

As explained before, we must start by reading the requirement specification, which is presented here. Furthermore, object-oriented analysis and design are performed in an iterative process using class diagrams. Analysis is performed on a piece of the system, design details are added to this partial analysis model, and then the design is implemented. Changes can be made to the implementation and brought back into the analysis model to continue the cycle. This iterative process is unlike the traditional waterfall technique, in which all analysis is completed before design begins.

8.5.1 Identifying Classes' Relationships

One of the strengths of object-oriented analysis is the ability to model objects as they exist in the real world. To accurately do this, you must be able to model more than just an object's internal workings. You also must be able to model how objects relate to each other. Several different relationships exist in the ViaNet bank ATM system, so we need to define them.

8.5.2 Developing a UML Class Diagram Based on the Use-Case Analysis

The UML class diagram is the main static analysis and design diagram of a system. The analysis generally consists of the following class diagrams:

- One class diagram for the system, which shows the identity and definition of classes in the system, their interrelationships, and various packages containing groupings of classes.
Multiple class diagrams that represent various pieces, or views, of the system class diagram.

Multiple class diagrams, that show the specific static relationships between various classes.

First, we need to create the classes that have been identified in the previous chapter; we will add relationships later (see Figure 8–8).

### 8.5.3 Defining Association Relationships

Identifying association begins by analyzing the interactions of each class. Remember that any dependency between two or more classes is an association. The following are general guidelines for identifying the tentative associations, as explained in this chapter:

- Association often corresponds to verb or prepositional phrases, such as *part of*, *next to*, *works for*, or *contained in*.
- A reference from one class to another is an association. Some associations are implicit or taken from general knowledge.

Some common patterns of associations are these:

- **Location association.** For example, *next to*, *part of*, *contained in* (notice that a part-of relation is a special type of association).
- **Directed actions association.**
- **Communication association.** For example, *talk to*, *order from*.

The first obvious relation is that each account belongs to a bank client since each BankClient has an account. Therefore, there is an association between the BankClient and Account classes. We need to establish cardinality among these
classes. By default, in most CASE tools such as SA/Object Architect, all associations are considered one to one (one client can have only one account and vice versa). However, since each BankClient can have one or two accounts (see Chapter 6), we need to change the cardinality of the association (see Figure 8–9). Other associations and their cardinalities are defined in Table 8–1 and demonstrated in Figure 8–10.

### 8.5.4 Defining Super-Sub Relationships

Let us review the guidelines for identifying super-sub relationships:

- **Top-down.** Look for noun phrases composed of various adjectives in the class name.
- **Bottom-up.** Look for classes with similar attributes or methods. In most cases, you can group them by moving the common attributes and methods to an abstract class.
- **Reusability.** Move attributes and behaviors (methods) as high as possible in the hierarchy.
- **Multiple inheritance.** Avoid excessive use of multiple inheritance.

CheckingAccount and SavingsAccount both are types of accounts. They can be defined as *specializations* of the Account class. When implemented, the Account

---

**TABLE 8–1**

<table>
<thead>
<tr>
<th>Class</th>
<th>Related class</th>
<th>Association name</th>
<th>Cardinality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account</td>
<td>BankClient</td>
<td>Has</td>
<td>One</td>
</tr>
<tr>
<td>Account</td>
<td>Account</td>
<td></td>
<td>One or two</td>
</tr>
<tr>
<td>SavingsAccount</td>
<td>CheckingAccount</td>
<td>Savings-Checking</td>
<td>One</td>
</tr>
<tr>
<td>CheckingAccount</td>
<td>SavingsAccount</td>
<td></td>
<td>Zero or one</td>
</tr>
<tr>
<td>Account</td>
<td>Transaction</td>
<td>Account-Transaction</td>
<td>Zero or more</td>
</tr>
<tr>
<td>Transaction</td>
<td>Account</td>
<td></td>
<td>One</td>
</tr>
</tbody>
</table>
class will define attributes and services common to all kinds of accounts, with CheckingAccount and SavingsAccount each defining methods that make them more specialized. Figure 8–11 depicts the super-sub relationships among Accounts, SavingsAccount, and CheckingAccount.

8.5.5 Identifying the Aggregation/a-Part-of Relationship
To identify a-part-of structures, we look for the following clues:

- **Assembly.** A physical whole is constructed from physical parts.
- **Container.** A physical whole encompasses but is not constructed from physical parts.
- **Collection-Member.** A conceptual whole encompasses parts that may be physical or conceptual.
A bank consists of ATM machines, accounts, buildings, employees, and so forth. However, since buildings and employees are outside the domain of this application, we define the Bank class as an aggregation of ATMMachine and Account classes. Aggregation is a special type of association. Figure 8-12 depicts the association, generalization, and aggregation among the bank systems classes. If you are wondering what is the relationship between the BankClient and ATMMachine, it is an interface. Identifying a class interface is a design activity of object-oriented system development; we look at defining the interface relationship in Chapter 12.

8.6 CLASS RESPONSIBILITY: IDENTIFYING ATTRIBUTES AND METHODS

Identifying attributes and methods is like finding classes, still a difficult activity and an iterative process. Once again use cases and other UML diagrams will be our guide for identifying attributes, methods, and relationships among classes.

Responsibilities identify problems to be solved. Beck and Cunningham explain this point elegantly [1, p. 2]. “A responsibility serves as a handle for discussing potential solutions. The responsibilities of an object are expressed by a handful of short verb phrases, each containing an active verb. The more that can be expressed by these phrases, the more powerful and concise the design.”

Attributes are things an object must remember such as color, cost, and manufacturer. Identifying attributes of a system’s classes starts with understanding the system’s responsibilities. We saw that a system’s responsibilities can be identified by developing use cases and the desired characteristics of the applications, such as determining what information users need from the system.

The following questions help in identifying the responsibilities of classes and deciding what data elements to keep track of [8]:

• What information about an object should we keep track of?
• What services must a class provide?

Answering the first question will help us identify attributes of a class. Answering the second question allows us to identify a class’s methods. Wirfs-Brock, Wilkerson, and Wiener describe system responsibility thus:

Responsibilities are meant to convey a sense of the purpose of an object and its place in the system. The responsibilities of an object are all the services it provides for all the contracts it supports. When you assign responsibilities to a class, you are stating that each and every instance of that class will have those responsibilities, whether there is just one instance or many. [8, p. 62]

In the following sections, we look at guidelines for identifying attributes and methods of classes in the problem domain by analyzing use cases. Furthermore, developing other UML diagrams such as UML activity and state diagrams also can assist in this process by helping us better understand classes’ responsibilities.

8.7 CLASS RESPONSIBILITY: DEFINING ATTRIBUTES BY ANALYZING USE CASES AND OTHER UML DIAGRAMS

Attributes can be derived from scenario testing; therefore, by analyzing the use cases and sequence/collaboration, activity, and state diagrams, you can begin to understand classes’ responsibilities and how they must interact to perform their tasks. The main goal here is to understand what the class is responsible for knowing. Responsibility is the key issue. Imagine yourself as an object in an object-oriented environment; what kind of questions would you like to ask [3]?

• How am I going to be used?
• How am I going to collaborate with other classes?
• How am I described in the context of this system’s responsibility?
• What do I need to know?
• What state information do I need to remember over time?
• What states can I be in?

Using previous object-oriented analysis results or analysis patterns (if these are available) can be extremely useful in finding out what attributes can be reused directly and what lessons can be learned for defining attributes [3]. Furthermore, you can start to extrapolate which classes you will have to build and which existing classes you can reuse. As you do this, you also begin thinking about the inheritance structure. If you have several classes that seem related but have specific differences, you probably want to make them common subclasses of an existing class or one that you define. Often, bottom up, the superclasses are generated while coding, as you realize that common characteristics can be factored out or in.

8.7.1 Guidelines for Defining Attributes

Here are guidelines for identifying attributes of classes in use cases:

• Attributes usually correspond to nouns followed by prepositional phrases such as cost of the soup. Attributes also may correspond to adjectives or adverbs.
- Keep the class simple; state only enough attributes to define the object state.
- Attributes are less likely to be fully described in the problem statement. You must draw on your knowledge of the application domain and the real world to find them.
- Omit derived attributes. For example, do not use time elapsed since order. This can be derived from time of the order. Derived attributes should be expressed as a method.
- Do not carry discovery of attributes to excess. You can add more attributes in subsequent iterations.

Another point to remember is that you may think of many attributes that can be associated with a class. You must be careful to add only those attributes necessary to the design at hand. Let use cases guide you in this process. For example, your initial thought is that the library Member class may have attributes such as Name, Social Security Number, Age, and Weight. The attributes Age and Weight may be important to the class Member in a personal system, but it is not within the scope of the system since there is no scenario in Library Borrow Books that requires or needs to keep track of the Age and Weight of the member.

8.8 DEFINING ATTRIBUTES FOR VIANET BANK OBJECTS

In this section, we go through the bank system classes and define their attributes.

8.8.1 Defining Attributes for the BankClient Class

By analyzing the use cases, the sequence/collaboration diagrams (see Chapter 7) and the activity diagram (see Chapter 6), it is apparent that, for the BankClient class, the problem domain and system dictate certain attributes. In essence, what does the system need to know about the BankClient?

By looking at the activity diagram (see Figure 6–9) we notice that the BankClient must have a PIN number (or password) and Card number. Therefore, the PIN number and cardNumber are appropriate attributes for the BankClient. Let’s now take a look at the use cases in Chapter 6 (Figures 6–9, 6–10, and 6–11) and Chapter 7, section 7.6. Other attributes of the BankClient class are drawn from our general knowledge of a BankClient. This usually is the case for defining attributes. The attributes of the BankClient are

```java
firstName
lastName
pinNumber
cardNumber
account: Account
```

At this stage of the design we are concerned with the functionality of the BankClient object and not with implementation attributes.

8.8.2 Defining Attributes for the Account Class

Similarly, what information does the system need to know about an account? Based on the use cases in Chapter 6 (see Figures 6–9, 6–10, and 6–11) and the
sequence/collaboration diagrams in Chapter 7 (see section 7.6 and Figures 7–4, 7–5, 7–6, 7–7, and 7–8), BankClient can interact with its account by entering the account number and then could deposit money, get an account history, or get the balance. Therefore, we have defined the following attributes for the Account class:

number
balance

8.8.3 Defining Attributes for the Transaction Class

The Transaction class, for the most part, must keep track of the time and amount of a transaction. Here are the attributes for the Transaction class:

transID
transDate
transTime
transType
amount
postBalance

8.8.4 Defining Attributes for the ATMMachine Class

Recall from Chapter 7 that the ATMMachine class was identified as part of the common class pattern (Tangible Things and Devices), which are physical objects or groups of objects that are tangible and with which the application interacts. Therefore, most attributes for this class describe its physical location and its state. The ATMMachine class could have the following attributes:

address
state

8.9 OBJECT RESPONSIBILITY: METHODS AND MESSAGES

Objects not only describe abstract data but also must provide some services. Methods and messages are the workhorses of object-oriented systems. In an object-oriented environment, every piece of data, or object, is surrounded by a rich set of routines called methods. These methods do everything from printing the object to initializing its variables.

Every class is responsible for storing certain information from the domain knowledge. It also is logical to assign the responsibility for performing any operation necessary on that information. By the same token, if an object requires certain information to perform some operation for which it is responsible, it is logical to assign it the responsibility for maintaining the information [8].

Operations (methods or behavior) in the object-oriented system usually correspond to queries about attributes (and sometimes association) of the objects [7]. In other words, methods are responsible for managing the value of attributes such as query, updating, reading, and writing; for example, an operation like getBalance, which can return the value of an account's balance. In the same fashion, we need
a set of operations that can maintain or change values; for example, an operation like setBalance to set the value of the balance.

In this section, we learn how to define methods based on the UML diagrams, such as statechart, activity, and sequence/collaboration diagrams and use cases.

8.9.1 Defining Methods by Analyzing UML Diagrams and Use Cases

In Chapter 7, we learned that, in a sequence diagram, the objects involved are drawn on the diagram as vertical dashed lines. Furthermore, the events that occur between objects are drawn between the vertical object lines. An event is considered to be an action that transmits information. In other words, these actions are operations that the objects must perform and, as in the attributes, methods also can be derived from scenario testing.

For example, to define methods for the Account class, we look at sequence diagrams for the followings use cases (see Chapter 7):

- Deposit Checking
- Deposit Savings
- Withdraw Checking
- Withdraw More from Checking
- Withdraw Savings
- Withdraw Savings Denied
- Checking Transaction History
- Savings Transaction History

Sequence diagrams can assist us in defining the services the objects must provide. For example, by studying the sequence diagram for Withdraw Checking (see Figure 7–5), it is clear that the Account class (which is the superclass of CheckingAccount and SavingsAccount) must provide a service such as withdrawal. By analyzing the use cases, such as the one in Figure 6–11, it is apparent that Account class should provide the deposit operation. These behaviors are defined as services of the classes in the business model. Ultimately, these services are implemented as the methods for your objects.

8.10 DEFINING METHODS FOR VIANET BANK OBJECTS

Operations (methods or behavior) in the object-oriented system usually correspond to events or actions that transmit information in the sequence diagram or queries about attributes (and sometimes associations) of the objects [7]. In other words, methods are responsible for managing the value of attributes such as query, updating, reading, and writing.

8.10.1 Defining Account Class Operations

Deposit and withdrawal operations are available to the Client through the bank application, but they are provided as services by the Account class, since the account objects must be able to manipulate their internal attributes (that is, modify the bal-
ance based on the transaction). Account objects also must be able to create transaction records of any deposit or withdrawal they perform.

Here are the methods that we need to define for the Account class:

deposit
withdraw
createTransaction

The services added to the Account class are those that apply to all subclasses of Account; namely, CheckingAccount and SavingsAccount. The subclass will either inherit these generic services without change or enhance them to suit their own needs. For example, we will override the withdraw method of the CheckingAccount class.

8.10.2 Defining BankClient Class Operations
Analyzing the sequence diagram in Figure 7-4, it is apparent that the BankClient requires a method to validate clients’ passwords (see Figure 8-13).

8.10.3 Defining CheckingAccount Class Operations
The requirement specification states that, when a checking account has insufficient funds to cover a withdrawal, it must try to withdraw the insufficient amount from

---

**FIGURE 8-13**
A more complete UML class diagram of the ViaNet bank ATM system.
its related savings account. To provide the service, the CheckingAccount class needs a withdrawal service that enables the transfer. Similarly, we must add the withdrawal service to the CheckingAccount class. The withdrawal service appears in the CheckingAccount class symbol (see Figure 8–13).

**8.11 SUMMARY**

The chapter describes the guidelines for identifying object relationships, attributes, and methods. Identifying relationships among objects is important since these interactions and relationships become the application. We look at three types of relationships: association, super-sub structure (generalization hierarchy), and aggregation or a-part-of relations.

Association is a relationship among classes. The hierarchical relation allows the sharing of properties or inheritance. The a-part-of structure provides the means to organize components of a bigger object.

To identify associations, begin by analyzing the interactions of each class and responsibilities for dependencies. Look for a dependency between two or more classes; it is a hint that an association exists. Associations often correspond to verb or prepositional phrases, such as *part of*, *next to*, *works for*, or *contained in*. Furthermore, a reference from one class to another is an association. Some associations are implicit or taken from general knowledge. Some common associations patterns are *next to*, *part of*, and *contained in* a relation; directed actions and communication associations include *talk to* or *order from*.

To identify super-sub relationships in the application, look for noun phrases composed of various adjectives in the class name in top-down analysis. Specialize only when the subclasses have significant behavior. In bottom-up analysis, look for classes with similar attributes or methods. Group the classes by moving those classes with common attributes and methods as high as possible in the hierarchy. At the same time, do not create very specialized classes at the top of the hierarchy. This balancing act can be achieved through several iterations. The process ensures that you design objects that can be reused in another application. Finally, avoid excessive use of multiple inheritance. It is more difficult to understand programs written in a multiple inheritance system. One way of achieving the benefits of multiple inheritance is to inherit from the most appropriate class and add an object of another class as an attribute.

The a-part-of relationship, sometimes called *aggregation*, represents a situation where a class comprises several component classes. A class composed of other classes does not behave like its parts but very differently. For example, a car consists of many other classes, one of which is a radio, but a car does not behave like a radio. Some common aggregation/a-part-of patterns are assembly, container, and collection-member. The a-part-of structure is a special form of association, and similarly, association can be represented by the a-part-of relation.

Identifying attributes and methods is like finding classes, a difficult activity and an iterative process. Once again, the use cases and other UML diagrams will be a guide for identifying attributes, methods, and relationships among classes.
Methods and messages are the workhorses of object-oriented systems. The sequence diagrams can assist us in defining services that the objects must provide. An event is considered to be an action that transmits information; therefore, these actions are operations that the objects must perform. Additionally, operations (methods or behavior) in the object-oriented system usually correspond to queries about attributes and associations of the objects. Therefore, methods are responsible for managing the value of attributes such as query, updating, reading, and writing.

**KEY TERMS**
- Aggregation (p. 182)
- Antisymmetry (p. 182)
- A-part-of relation (p. 182)
- Assembly (p. 183)
- Association (p. 178)
- Collection-member (p. 184)
- Container (p. 183)
- Directed actions associations (p. 180)
- Ternary association (p. 180)
- Transitivity (p. 182)

**REVIEW QUESTIONS**
1. Why is identifying class hierarchy important in object-oriented analysis?
2. What is association?
3. What is generalization?
4. How would you identify a super-subclass structure?
5. What is an a-part-of structure? What are major properties of an a-part-of structure?
6. What guidelines would you use to identify a-part-of structures?
7. Is association different from an a-part-of relation?
8. What are some common associations?
9. What are unnecessary associations? How would you know?
10. Why do we need to identify the system's responsibilities?
11. How would you identify attributes?
12. How would you identify methods?
13. What are unnecessary attributes?
14. What does repeating attributes indicate?
15. Why do we need to justify classes with one attribute?

**PROBLEMS**
1. Do a literature search on object-oriented analysis patterns and write a report based on your findings.
2. See the details regarding Grandma's Soups Unlimited in Chapter 6:
   a. Identify a super-subclass relationship by following the guidelines for generalization.
   b. Identify an a-part-of structure by following the guidelines for an a-part-of structure.
c. Identify association for the classes in the problem by following the guidelines for identifying relationships and methods.

d. Identify attributes and attributes for the classes in the Grandma’s Soups Unlimited problem, by following the guidelines for identifying methods and attributes.

3. Identify some of the attributes and methods in Grandma’s Soups Unlimited (see Chapter 6).

REFERENCES

1. Beck, Kent; and Cunningham, Ward. “A Laboratory for Teaching Object-Oriented Thinking.” Object-Oriented Programming System Languages and Application. OOPSLA’89, October 1-6, 1989, New Orleans, LA.


