Chapter Twelve

Object-Oriented Software Development

Analysis and Design

Summing Up

You have seen several class definitions, modified some classes, implemented class member functions (in programming projects for Chapters 6 through 11), and perhaps completely implemented a few classes (in programming project 8H, "The `elevator` Class"; 10K, "The `stats` Class"; or 11E, "The `set` Class"). All of these previous activities relate to the implementation of classes.

Coming Up

This chapter discusses another facet of the object-oriented paradigm—the analysis and design of a system with objects as the main building blocks.

You will study an object-oriented software development strategy in the context of a real-world problem—building a jukebox system for a campus student center. The analysis in this chapter will help the programming team design the class definitions and implement the member functions in Chapter 13. After studying this chapter, you will be able to

- identify classes that model a solution to a problem
- assign responsibilities to classes
- use Component/Responsibility/Helper cards to aid analysis and design
- understand the need for analysis and design before implementation

  Analysis and design projects to reinforce object-oriented development

12.1 Object-Oriented Analysis

So far, we have only considered individual classes and some low-level design issues, such as where the data members should go in a C++ class, why constructors are needed, and member function implementations. However, before individual classes are designed and implemented, they are first recognized as part of some larger software system.

This section introduces an object-oriented design (OOD) methodology in the context of a real-world problem—the cashless jukebox. The strategy is based on the responsibility-driven design methodology of Wirfs-Brock, Wilkerson, and Wiener [Wirfs-Brock 90].

Another major component of object-oriented software development is the Component/Responsibility/Collaborator (CRC) card introduced by Kent Beck and Ward Cunningham [Beck/Cunningham 89]. This simple and effective tool consists of a set of 3-by-5-inch index cards. CRC cards were first used to help people understand object technology. Developers employ them to develop large-scale, object-oriented systems. Because the word *collaborator* has caused confusion, CRC cards will henceforth be referred to as Component/Responsibility/Helper (CRH) cards.

The first step in object-oriented analysis involves identification of the *key abstractions*—the major classes. The system is modeled as a set of classes, each with its own responsibilities. You will see how team members "become" an object through role playing. Playing the roles of the objects helps the team identify the responsibilities of each while determining the relationships between them. Role playing helps the team understand the problem, model a solution, and iron out the rough spots early in development, when it is relatively cheap and easy to do so.

Self-Check

To warm up to an object-oriented view of systems, first answer each of the following "Pre-Check" questions. Look at the answer for each before going on to the next. These "Pre-Check" questions are intended to get you to understand the essentials of planning the design of some larger systems.

12-1 On one piece of paper, sketch the design of a small one-branch bank such that someone unfamiliar with the system would understand the major components, the key purpose of each, and any interaction between those components.

12-2 On one piece of paper, sketch the design of a small one-branch library such that someone unfamiliar with the system would understand the major components, the key purpose of each, and any interaction between those components.

12.1.1 The Problem Specification

In the spring of 1997, the author suggested building a CD music jukebox using a 200-capacity compact-disc player. Sophomore Ed Slatt, a computer engineering major, guaranteed that he would figure some way to control that CD player from a computer. By the end of the semester, Ed had a prototype infrared remote (IR) unit working. He used a circuit design that Chris Dodge has made available on the Web.[1] His `IRDEV` class and some assembler code were wrapped up in a `cdPlayer` class. This allowed messages like these that play the fifth track on the CD in the third tray of a seven-disc CD player:

```
// Construct an object that can send signals to a real CD player
cdPlayer myCdPlayer;

// Play the fifth track in the third tray of the physical CD player
myCdPlayer.play(3, 5);
```

Ed then went on to pursue his computer engineering degree.

In the fall of 1997, several students asked the same author to provide an honors option for the first course in the computer science major. They decided to build the jukebox. The events you are about to read are based on this true story. The following problem statement begins the project:

1. At the time of this writing, Chris Dodge has the information located at this URL: **http://www.ee.washington.edu/eeca/circuits/PCIR/Welcome.html**

Problem Statement for the Cashless Jukebox

The Student Affairs office has decided to put some new-found activity fee funds toward a music jukebox in the student center. The jukebox will allow a student to play individual songs. No money will be required. Instead, a student will swipe a magnetic student ID card through a card reader. Students will each be allowed to play up to 1,500 minutes worth of "free" jukebox music in their academic careers at the school, but never more than two selections on any given date. A student may select a CD from the available collection of CDs and then an individual song from that CD. Students may not play entire CDs.

This project is now assigned to a team that includes Blaine, Director of Student Affairs. Blaine is playing the role of the *domain expert*—the person who knows about student policies. His office is in the student center, not far from where the jukebox might be installed. Blaine also represents the *client*—the one with the money. The team will also have an *object expert*—Chelsea—to help guide the team though this, their first try at object-oriented software development. Chelsea's experience and knowledge of object-oriented design heuristics will likely help the team make better design decisions. The team has two chemical engineering majors—Jessica and Jason—who are taking an honors option in the middle of a first programming course. Also on the team are Matt, a computer engineering major, and Steve, an engineering science major, who are both taking the introductory course. The team also has Charlie, a second-year computer science major who promised to help Matt implement the jukebox (see Chapter 13). Charlie has more programming experience than any of the other team members. The team is completed by Misty, who is deciding between a math or a computer science career.

At the first meeting, the team decides to hammer out an idea of what this cashless jukebox should be able to do. During analysis, the team will be concerned with *what* the system must do. The team will not worry too much over the *how.* During analysis, the terminology and activities relate to those who requested the software. The analysts should be able to communicate their thoughts with those who requested the software. The analyst must also

- work with the clients to refine requirements
- challenge the requirements
- probe for missing information

12.1.2 The Goal of the Analysis Phase

The goal of the analysis phase is to create an abstract model in the vocabulary of the client (the Student Affairs office, in this case). This can be accomplished by way of the following three-step strategy:

1. Identify classes that model (shape) the system as a natural and sensible set of abstractions.
2. Determine the purpose, or main responsibility, of each class. The responsibilities of a class are what an instance of the class must be able to do (member functions) and what it must know about itself (data members).
3. Determine the helper classes for each. To help complete its responsibilities, a class typically delegates responsibility to one or more other objects. These are called *helpers*.

Before the team begins to analyze the problem, Chelsea tells them that software developers do not immediately find all classes. It is also unlikely that all responsibilities will be discovered in the first pass. Some of the helper classes may also be missed. It is perfectly acceptable to change the list and names of the classes that shape the solution as the problem is considered over time. Also, new classes may be required when moving into the design and implementation phases.

For now, don't worry about creating the perfect model. There are many false starts. And because analysis sometimes recognizes uncertainty in a problem statement, there may be some new concerns that need to be settled. The team will be trying to understand the requested software system at the level of those requesting the software.

Also realize that there is rarely one true correct best design. Relax. Tell yourself now that all designs are valid. So don't be afraid to make mistakes. It is much easier to change things during analysis and design than after the system has been deployed.

12.1.3 Identification of Classes that Model the Solution

The first goal, or deliverable, in object-oriented software development is a set of classes that potentially model the system. Each class will be assigned its major responsibility.

One simple tool for getting started is to write down all the reasonable nouns and noun phrases in the problem statement. Consider these as potential classes representing part of the solution. You may also record potential classes even if they were not written as nouns in the problem statement. For example, someone on the team with expertise in the problem area may suggest useful classes due to his or her understanding of the domain—in this case, students and music. Potential classes may come from words that are spoken while describing the system or questioning the problem statement. In summary, useful classes for modeling the system may come from sources such as these:

- the problem statement
- an understanding of the problem domain (knowledge of the system that the problem statement may have missed or taken for granted)
- the words floating around in the air during analysis of the problem

And while considering potential classes, consider the following object-oriented design heuristic:

Object-Oriented Design Heuristic 12.1 (Riel's 3.6)

Model the real world whenever possible.

This design heuristic leads to more understandable software. It not only helps during analysis, but also during maintenance when someone unfamiliar with the system must fix a bug, add an enhancement, or update the software to match real-world changes.

The team established the following list of noun phrases from the problem statement (redundant entries were not recorded).

Noun Phrases—All Nouns in the Problem Statement

Student Affairs office	minutes	student ID card	student
activity fee funds	date	academic careers	money
music jukebox	card reader	student center	CD
collection of CDs	selection	jukebox music	song

The noun phrases of a problem statement fall into three categories that indicate their potential viability as classes to model a solution.

1. somewhat sure
2. not sure
3. should not be considered—irrelevant or has the propensity to muddle

There are many guidelines intended to help us discover useful classes. One guideline has already been used above—eliminate redundant entries. There was no useful purpose for writing down "student" three times, for example. Only consider the noun phrases that have meaning in the realm of the system. The activity fee funds provide the money to pay for the jukebox; however, that money will not be part of the design.

The following set of noun phrases represents a first attempt to identify the key abstractions in the problem statement (irrelevant nouns omitted).

Potential Classes—A First Pass at Finding Key Abstractions

Somewhat Sure	Not Sure
jukebox	activity fee funds
student	money
song	jukebox music
card reader	date
student ID card	
CD	
collection of CDs	

Self-Check

12-3 Write a list of potential classes after reading the following problem statement: (*Note:* This problem specification will be used in later self-check questions.)

The college library has requested a system that supports a small set of library operations. Students may borrow books, return those borrowed books, and pay fees. The late fee and due date have been established as follows:

	Late Fee	Borrowing Period
Books:	$0.50 per day	14 days

The due date is set when a borrowed item is checked out. A student may borrow only seven books. Any student with seven books checked out may not borrow anything more.

The team considers each potential class in more detail for help in understanding the system.

12.1.3.1 Jukebox? YES, coordinates all activities

The programmers on the team feel that `jukeBox` seems an appropriate name for the class that will be responsible for coordinating activities such as handling requests from the users of this system. There might be one instance of `jukeBox` in the program that gets things going and keeps things going.

12.1.3.2 Student? YES, maintains student information

It is suggested that a class named `student` will prove itself a useful abstraction for modeling the users. After all, students will be involved in playing music. Chelsea advises the team to distinguish between the human version of the student that approaches the jukebox with a student ID card and the software model of that student stored inside the computer. A friendly argument ensues. Although no one seems to recognize it at first, the confusion is due to the name `student`. Steve suggests that the team employ the name "user" to indicate the physical student that uses the jukebox. The programmers agree to keep `student` as the name of the class that models that physical real-world user. `student` is the software equivalent of "user."

12.1.3.3 Song? NO; Track? YES, one of the tracks on a CD

Because students are allowed a certain amount of play time in minutes, it appears that `song` would be a useful class. Each `song` object should know its playing time in minutes and seconds along with its title. Jason and Jessica see this as an instance of the State Object pattern they learned about earlier in their first computer science course.

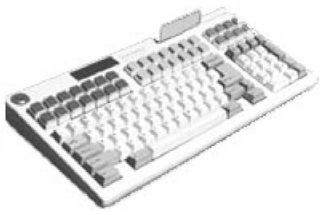
Matt changes the subject by claiming that there are some CDs that put two songs into one track. Other CDs have some blank air time at the end. Blaine observes, "Rock and roll has changed since I was young." He can't believe what he is hearing. Additionally, Charlie relates that he has a Beethoven CD where one "song" (symphony, actually) is distributed over several tracks. Each track is a movement—the first movement, the second movement, and so on.

Chelsea indicates that the team can better communicate the design if the name is changed to `track`. The user will select a `track` from a CD. That sounds better.

12.1.3.4 Card reader? YES, the object used to read the magnetic student ID cards

The card reader is one of several physical objects in the cashless jukebox. Should it be a class? No one is sure. Ed suggests that magnetic card readers are easy to come by. It may be a useful abstraction. After a quick check on the Web, Jessica produces this picture of a combination keyboard and magnetic card reader to solidify the concept of a card reader.

Figure 12.1. *Magnetic card reader keyboard*



Picture courtesy of B & C Data Systems, Gorham, Maine

Although the physical card reader exists, it will still be useful to have a software abstraction for this physical entity—analogous to the user/`student` relationship. The team decides to use the class name `cardReader` for the object that gets input from magnetic ID cards.

12.1.3.5 Student ID card? NO, the object inserted into the magnetic card reader

The student ID card is one of several physical objects in the cashless jukebox. However, the team decides it is "outside" the system. Although it allows students to gain access to the jukebox, it need not be modeled. It is already done. Every student has a student ID card.

Chelsea congratulates the team as she tells them about a widely accepted object-oriented design heuristic:

Object-Oriented Design Heuristic 12.2

Eliminate classes that are outside the system.

According to Arthur Riel [Riel 95], the hallmark of such a class is one whose only importance to the system is the data contained in it. While the student identification number might be of great importance, the system should not care whether the ID number was read from a swiped magnetic ID card, typed in at the keyboard, or "if a squirrel arrived carrying it in his mouth."

12.1.3.6 CD? YES, each `CD` object stores a collection of `tracks`

Someone on the team doesn't agree that `CD` should be a key abstraction. Jason, one of the computer science majors, chimes in and suggests, "This is not a problem, I can visualize a `CD` object as a `vector` of `tracks`." Blaine asks, "What is a vector?" Chelsea tries to rectify unfamiliar and differing terminology with some anthropomorphic object-speak: "A `CD` is responsible for knowing what `tracks` it has." Blaine complains that Chelsea is trying to give human qualities to the CD. Jessica responds that this is precisely what anthropomorphism means. She also heard in her CmpSc class that giving human qualities to objects happens quite often during object-oriented software development. It really helps.

Steve still has a problem with the singularity of a CD and a song: "Certainly the system must maintain many `CD` objects, each of which may store many songs. How can we talk about just one class when there are many songs on so many CDs?" Jason retorts that one class is used to create many instances, or objects. And right now, the team should concentrate on identifying classes to model the real-world components that are part of a jukebox.

Steve agrees that there should be more than one CD for students to choose. Jason relates how they have just learned about two different container classes for storing collections of objects—`vector`s and `bag`s. "Bags and vectors?" wonders Blaine. Jessica tells Blaine that he needn't worry about these implementation details and reminds Jason that the programming team can choose the appropriate container class later. Instead, Jessica introduces the notion of a class that is responsible for storing all the CDs. Steve asks if we could rename this `cdCollection`. The team unanimously approves.

12.1.3.7 Collection of CDs? YES, all CDs that could be listened to

Jason is thinking that behind the public interface of the `cdCollection` class, the data members may include a `vector` of `CD` objects and the number of CDs physically stored in the CD player. Matt suggests that someone will have to make sure the software version is always in sync with the physical CD collection. Chelsea encourages the team to postpone these implementation details for now.

12.1.4 Any Other Classes?

Steve and Matt launch into a discussion about who is allowed to play a track. Shouldn't a collection of students also be maintained? Jason agrees, but he complains it may be difficult to keep a collection of all valid students. Should the jukebox allow anyone in the world with a magnetic ID card to play a track? How might the system prevent unauthorized access? It appears certain that some part of the cashless jukebox will have to maintain a collection of students. Although the noun phrase "collection of students" does not exist in the problem statement, Jessica says that it helps her to think of a `studentCollection` class that is responsible for maintaining the time credit available for each student. Chelsea recommends the team go with their instincts and add a `studentCollection` class to the design.

12.1.4.1 `studentCollection` stores a list of students who could select songs

Of the original set of potential classes, all but one survived the first cut at modeling the system. Two new key abstractions (classes) were added. One is for storing the collection of CDs. The other maintains the list of students who are allowed to use the jukebox. However, an uneasy feeling pervades the team. Misty thinks there is something missing. The team reviews the classes. Ed suggests that there must be some object that actually plays the selected song.

Jessica suggests a solution. The team could purchase an old jukebox and modify it to allow card swipes. However, this would not prevent students from playing more than two songs on any given date. Ed says that a computer could be hooked up to the jukebox. Then the computer would be responsible for reading student ID cards and sending play messages to the physical jukebox. The physical jukebox has the responsibility of selecting the physical CD and playing the correct track. Commercially available jukeboxes do this. Ed states that another major advantage derives from the fact that jukeboxes also have built-in stereo systems. This could save money. Blaine reminds Steve that the student center already has a potent stereo system. Any money saved on this project could be funneled back into other student projects. So, as is done in the world of business, a decision must be made that relates to the bottom line.

Chelsea investigates on the Internet and finds a place to purchase a CD jukebox that holds 50 CDs. It looks very cool. However, it lists for $4,500. And this does not include the CDs. However, according to the Wurlitzer Jukebox Co. representative, they could probably modify their jukebox to allow magnetic ID card input.

FIGURE 12.2. *The Wurlitzer "One More Time" CD Jukebox*

Picture courtesy of Wurlitzer Jukebox Co., Pittsburgh, Pennsylvania

The team considers cost alternatives. Matt informs the team that the computer department has some personal computer components sitting around gathering dust. He suggests that for little or no money, he could put together a computer capable of implementing the appropriate classes. A card reader keyboard would have to be purchased no matter what.

Charlie suggests that a computer could be connected to a 200- or 300-disc CD player. The CD player in turn could easily be connected to the existing student center stereo system. Chelsea reminds the team, "Ed already did this." The team considers adding another object to model the system—a compact-disc player.

12.1.4.2 `cdPlayer` plays any track from any CD

The compact-disc player is yet another physical object in the cashless jukebox. But should it be a class? After all, several electronic stereo manufacturers already implement this physical device. Chelsea suggests the team try to understand the system using a common vocabulary. A CD player appears to be a very important part of the solution. The implementation would come later. The team decides to have a `cdPlayer` class to represent the responsibilities of this "real" object—analogous to the `student` and `cardReader` classes, which represent other real objects. The actual CD player has not been purchased yet. Jessica pulls a picture off the Web to help bring a more concrete feel to these abstractions.

FIGURE 12.3. *Sony CDP-CX200 CD player*

Picture courtesy of Sony Electronics Inc.

Now that the team has added a `cdPlayer` class, Misty suggests the team consider a stereo system class.

The stereo system certainly is part of the entire system. It is yet another one of those essential physical components. Chelsea asserts that we could consider the stereo system as part of the cashless jukebox. Its major responsibility is to take the output from the CD player and produce audible sounds at the right treble, bass, and volume settings. However, it is reasonable to set boundaries around the system under development. The team decides that sending a song selection message to `cdPlayer` represents such a boundary. The prebuilt electronic components will then take over and perform their well-defined responsibilities. Chelsea excitedly shouts out, "This is reuse!" The `cdPlayer` class is the interface to the music-generating part of this system. The team agrees that the stereo system is on the other side of the CD player, well outside the system being designed.

12.1.4.3 Stereo system? NO, amplifies the music

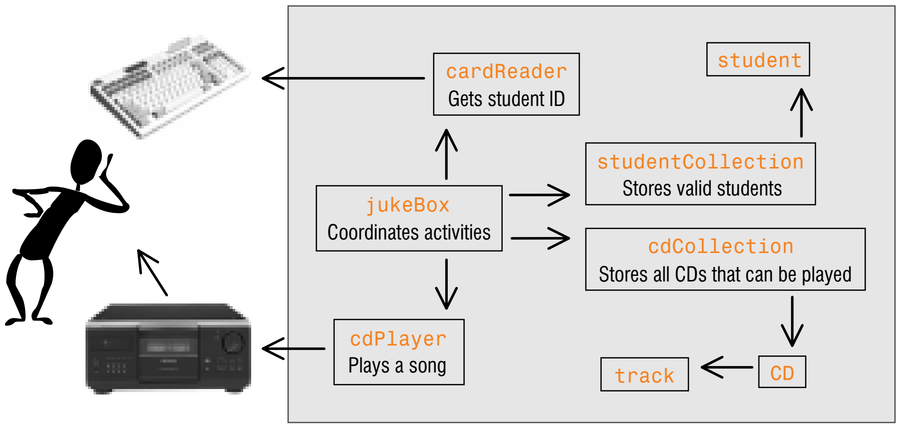
The team feels as though they have captured many key abstractions for this application. They have the beginning of a framework for analyzing the problem in more detail. There is a sense that the primary responsibility of each class has been recognized—or at least implied. Chelsea suggests that the class names be documented along with their major responsibilities.

Before splitting for pizza, the team documents the classes discovered during analysis with a sketch that includes

1. the class names
2. the primary responsibility of each class

3. arrows indicating possible message sends to helper classes
4. the boundaries of the software abstractions under development

FIGURE 12.4. *Analysis sketch of major classes that model a cashless jukebox (subject to change)*



Chelsea congratulates the team again. The design looks good. Out of the original eight noun phrases, one was eliminated (student ID card). The `studentCollection` class was discovered as the result of someone knowing something about collections of data and the necessity for searching for a particular student. Chelsea reports that such changes are typical. Writing the noun phrases is a tool to help find classes that effectively model the solution. It doesn't mean all nouns must become classes. It doesn't pretend to find them all. Charlie suggests that some of these classes may never even get implemented. Chelsea says that other new classes may be discovered as the team proceeds with role playing.

Self-Check

12-4 Reconsider the key abstractions (classes) for representing the college library. (See Self-Check 12-3.) Draw a picture like Figure 12.4 above that does the following:

1. lists all the classes that reasonably model the system
2. lists the major responsibility of each class
3. marks the boundary of the system—it's okay to show entities that are outside the system

12.2 Role Playing and CRH Card Development

The primary responsibility of each class has now been identified. The team will now set about the task of identifying and recording other responsibilities. Responsibility-driven design emphasizes identifying responsibilities and assigning those responsibilities to the most appropriate class. Chelsea, the object expert, suggests that the team should often ask questions like this:

- What is this class responsible for?
- What class is responsible for a particular action (member function) or specific knowledge (data member)?

The team will also be identifying helpers. These are classes that one class needs to help carry out its responsibility. The object-oriented approach views the running solution as a collection of objects in which each object does its own thing for the good of the whole. There should not be any all-powerful class that does everything.

The team should try to answer questions such as: What are the other responsibilities needed to model the solution? Which class should take on this particular responsibility? and What classes help another class fulfill its responsibility? Responsibilities convey the purpose of a class and its role in the system. These questions are more easily addressed if the team remembers that each instance of a class will be responsible for

1. the knowledge that any object of the class must maintain, and
2. the actions that any object of the class can perform.

The team should always be prepared to ask these two questions:

1. What should any object of the class know (knowledge)?
2. What should any object of the class be able to do (action)?

"It's that anthropomorphism thing again," moans Blaine, the domain expert from the Student Affairs office.

Assigning a responsibility to a class means that every instance of that class will have the same responsibility. This is true when there are many instances of the class. There will be many `CD` and `student` objects for example. It is also true when there may be only one instance of the class—`jukeBox` and `cdCollection` for example. Later on, during design and implementation, the actions may become public member functions of a class. The knowledge may become data members.

The responsibilities may be identified from several sources such as:

- the problem statement (specification)

- the classes
- the ideas that float around the room, especially during role playing

These responsibilities must eventually be assigned to the appropriate class, either to the classes already identified or to new classes if necessary. For example, whose responsibility is it to play one particular song? This responsibility might be shared between several classes. `jukeBox` may send a message to `studentCollection` to find out whether or not a user may select a song. `jukeBox` cannot do this by itself—it needs help from `studentCollection` and `student`. In situations like this, when class A needs the help of class B, class B is said to be a helper of class A.

The object expert suggests that the team begin to assign responsibilities to classes using the simple tool of role playing. During role playing, each team member assumes the role of a class to see what happens when a certain situation arises; for example, what happens when a certain student wants to select a certain track. But before this is done, the object expert suggests that the team use another simple low-cost tool for capturing the decisions made during role playing.

12.2.1 CRH Cards

At this point, Chelsea writes the heading "Class:" at the top of a 3-by-5-inch index card. Under this, she also writes the column headings "Responsibilities:" and "Helpers:". Immediately, an inexpensive index card has metamorphosed into a Component/Responsibility/Helper (CRH) card. A CRH card records a class name, the responsibilities of the class, and the other classes required to help the class fulfill those responsibilities—the helpers. Here is an example CRH card with some possible responsibilities and helpers as it may appear much later in the software development phase.

Figure 12.5. *CRH card showing what `cdCollection` might develop into*

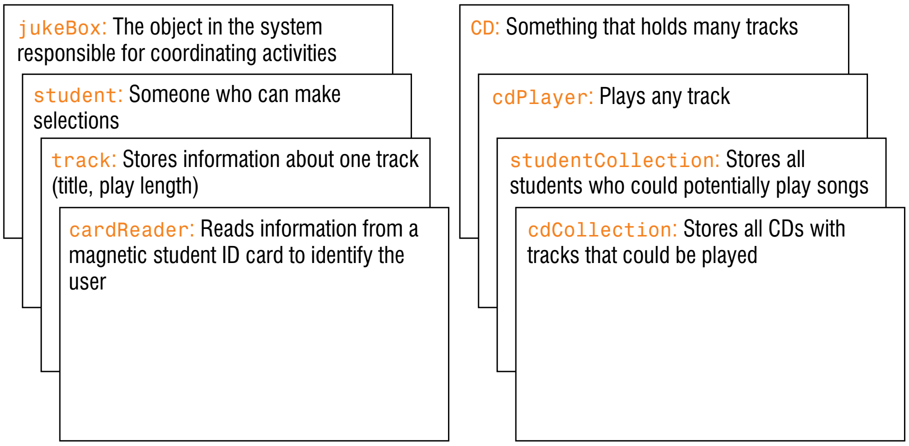
Class: `cdCollection`	
Responsibilities:	**Helpers:**
know all CDs	`vector` of CDs
retrieve a CD	
`addNewCD(CD aCD, int trayNumber)`	
`removeCD(int trayNumber)`	
`CD getCD(int trayNumber)`	

Although this CRH card for the `cdCollection` class contains return types and arguments, CRH cards typically list only the member function names (action responsibilities). Conversely, the CRH card might well indeed say "know all CDs" (knowledge responsibility). Knowledge responsibilities may later get turned into private data members such as `vector<CD> my_cds;`.

The responsibilities are what the object must be able to do and what the object must know. These responsibilities are written down on the front of the card under the heading "Responsibilities:". However, the previous CRH card represents design decisions yet to be made. It is only a preview of what it might look like.

Some CRH card practitioners use the backs of the CRH cards for recording different types of information. Chelsea suggests the team write down the class name along with its major responsibility on the back of the 3-by-5-inch index cards. Here is the result, using the design shown earlier.

Figure 12.6


jukeBox: The object in the system responsible for coordinating activities
student: Someone who can make selections
track: Stores information about one track (title, play length)
cardReader: Reads information from a magnetic student ID card to identify the user
CD: Something that holds many tracks
cdPlayer: Plays any track
studentCollection: Stores all students who could potentially play songs
cdCollection: Stores all CDs with tracks that could be played

Each team member takes one or two of these CRH cards to "become" the class. The holder of each card will play the role of that class.

12.2.2 Role Playing

Each team member has now assumed the role of at least one class (one team member could role play several classes if necessary). Chelsea suggests that the team role play a scenario to see how one instance of a class could interact with instances of other classes. Jessica wants to know, "What is a scenario?" Chelsea responds, "A scenario is the answer to the question, What happens when . . . ?" Steve suggests the team role play the following scenario.

12.2.2.1 Scenario 1: "What happens when a student wants to select a track?"

The first thing to do is decide which object begins the scenario. Each team member looks on his or her CRH card to see who will start. The team member playing the role of `jukeBox` states, "I'm playing the role of the object responsible for coordinating activities, so I guess I'll start."

`jukeBox`: I'm playing the role of the jukebox so I'll start this scenario. Hey `cardReader`, has a user swiped a card?

`cardReader`: No.

`jukeBox`: Hmmm. Okay, `cardReader`, has a student swiped a card?

`cardReader`: No.

`jukeBox`: Hey, how often do I have to do this? I suppose I'll have to wait for a user.

The object expert suggests that Jason (`jukeBox`) add a `waitForUser` responsibility to his card.

`jukeBox`: Okay, I'm waiting for a user. Waiting, waiting, waiting . . .

`cardReader`: Okay, `jukeBox`, a user just swiped a card. Here is all the information on that magnetic strip ID card.

`jukeBox`: Thanks, `cardReader`, but I don't know what to do with all of this stuff. Which part of the input uniquely identifies the student?

The team wonders what information is recorded on a magnetic student identification card. Is there a name, a student ID number, the amount of money left in the student's food service balance, an address? Since no one knows for sure, perhaps `cardReader` could take on the responsibility for getting the unique student identification number of the user holding the magnetic card just swiped.

`cardReader`: Okay, I'll add `getStudentID` to my list of responsibilities.

`jukeBox`: Thanks, `cardReader`. By the way, although I first believed you weren't necessary as a class, I do believe so now. You not only communicate with the physical card reader, you also have access to the entire student ID card so you can return the information I seek—the student ID number. You just made my life simpler. I don't need to know what kind of physical card reader is out there, nor do I have to know details such as the format of the magnetic card being swiped. Now that I have the student ID number, I need to verify that the user can select a song.

`studentCollection`: Well, since it is my responsibility to store the collection of student objects, let me check to see if there is a student here with that ID. Yes, here is the `student` object associated with the student ID number you passed to me. I will add a responsibility to the front of my CRH card: `getStudent`.

`jukeBox`: Now that I know the user is valid, I suppose we should let the user select a song, I mean `track`. Okay, whose responsibility is it to maintain the list of all tracks on all CDs?

`cdCollection`: That's me! What do you want?

`jukeBox`: I'd like the collection of all CDs.

`cdCollection`: Well, all you have to do is ask. I've been here all along.

`jukeBox`: Okay, I now know I always have access to `cdCollection`. What do I do with you now?

There is a pause. No one knows. Should `jukeBox` display the CD choices to the user and then, based on the CD selection, show the tracks? This would mean that `jukeBox` will have much more to do (Jason and Jessica are thinking about all those loops). Chelsea suggests that it seems like some other object should be responsible for getting this information from the user. `jukeBox` coordinates activities; it can delegate authority to other classes. It does not have to do everything.

Chelsea points out that there are many object-oriented designers who use the following design heuristic when confronted with similar design decisions:

Object-Oriented Design Heuristic 12.3

Avoid all-powerful (omnipotent) classes.

It is undesirable to have a class that does everything, or even too much. The by-product of assigning too much responsibility is a complicated class that can be difficult to implement and maintain after deployment into the student center. Chelsea describes another related object-oriented design heuristic that helps software developers attain well-designed object-oriented systems:

Object-Oriented Design Heuristic 12.4 (Riel's 3.1)

> Distribute system intelligence among the classes as evenly as possible. The top-level classes in a design should share the work uniformly.

Chelsea encourages the team to try to evenly distribute the work amongst the analysis classes that are currently being role played. The benefit is a more easily understood system. The design will be more easily communicated to the client and the programmers that will implement the design. It will be more easily understood when the inevitable bugs have to be fixed or enhancements are to be made.

`jukeBox`: Let me summarize: I won't try to get the user's track selection. I know that I will somehow have access to the CD collection, but what has to happen now? What should I do next to get the track?

Chelsea offers a suggestion: "A good object-oriented system has objects that delegate responsibilities to other objects. This is not about producing lazy objects. This is about understandable systems. Some other object could get the user selection." "But who?" asks Jason.

Chelsea proposes that to keep things simple, it might be appropriate to add a new class to interface with the user. The major responsibility would be `getTrack`. The team's computer engineer, Matt, and the computer science major Charlie, who are now quite familiar with interactive input and output, suggest that with a new `trackSelector` class, `jukeBox` need not worry about how to get output to and input from the user. It could come from the keyboard; from a touch screen; or through a graphical user interface for the Mac, a graphical user interface for a WinTel machine, a graphical user interface for a Unix or Linux system, or whatever. The `jukeBox` role player suggests backing up one step and finishing the current scenario with a new `trackSelector` class, which has the major responsibility of communicating with the user. `trackSelector` will get the user's selection.

12.2.2.2 `trackSelector`, a new class added to get student selections

`jukeBox`: Okay, I now have access to the collection of all CDs. I'll pass it to `trackSelector`. You tell me what track the user wants to hear.

`trackSelector`: Okay, I'll offer the options to the user and let him or her decide. Hey, what options are there? I'm in control here, but I still don't know how to get the user's selection.

`cdCollection`: I hold the collection of CDs, so it seems like I should have some mechanism to allow access to the CDs and all tracks on the CDs. Then you could look at every CD in my collection and show the artist and title, perhaps by artist name in alphabetic order. Once the CD is selected, you could iterate over all the tracks available on that CD.

`CD`: I think I can help. Those knowledge responsibilities are listed on my CRH card (know all tracks). However, I don't know how to reveal individual tracks. I will add that responsibility: "allow references to individual tracks."

"Yes, you could have a `vector` of `tracks` that are accessible with the subscript operator," asserts Jason. Chelsea warns Jason to avoid tying in implementation details at this point. He can decide later. For now it seems `cdCollection` must allow references to individual CDs and each CD must allow references to individual tracks. "Do you both have that responsibility on your individual CRH cards?" she asks.

`cdCollection`: No, I'll add the responsibility of allowing access to all CDs now. In fact, it seems like I will have to be able to add new CDs and delete others as the music in the physical CD jukebox changes. So I'll also add `addCD` and `removeCD`.

`CD`: I didn't list those responsibilities either, but I'll add them now. And in addition to knowing my play time, I could also be responsible for knowing my CD number and my track locations. This is all `cdPlayer` would need to know in order to play any track.

`trackSelector`: Look, you guys, if you are done for now, I'd like to just say that I can get the track selected by the user. Let's continue the scenario. Okay, `jukeBox`, here is the track selected by the user. I'll write the responsibility `getTrack` on my CRH card.

`jukeBox`: Thanks, `trackSelector`. Now I need to determine whether or not the current student can play the selected track. Who is responsible for knowing the play time of any track?

`track`: It seems like I should be responsible for knowing the duration of my track. For example, it might take 3 minutes and 34 seconds to be completely played. I'll add the operation "know play time" to my CRH card along with the responsibility to know the CD number and what track number I am.

"Whoooooaaaaa, this is confusing!" proclaims Blaine, the domain expert. "Why does `jukeBox` need to know the duration of the track?" Jessica suggests that some object must be made responsible for determining whether or not this particular student can play this particular song. Chelsea suggests that the team continue role playing until this scenario reaches some logical conclusion.

`jukeBox`: Okay, now I should check to make sure the student is able to select this track before telling `cdPlayer` to play it. I seem to remember I cannot simply play a track without checking on a few things. I know the current student and the selected track. What do I do now?

The team has stalled. They wonder what must be done next. Why did `jukeBox` need to know the student in the first place? Misty reminds the team that jukebox play time is reserved for this school's valid students only. Therefore `jukeBox` had to ask `studentCollection` to validate the user first. Well actually, `studentCollection` has that responsibility. The verification was done earlier in this scenario (even if it is not exactly clear yet how it was done). Additionally, as stated in the problem statement, a student must have enough time credit, and may not play more than two tracks on the same date. The role player of `track` has an idea. He suggests that the jukebox ask the track for its play time.

12.2.2.3 Alternative 1

`jukeBox`: So tell me, `track`, how many minutes and seconds do you require to be played?

`track`: 3 minutes and 34 seconds.

`jukeBox`: `student`, do you have at least 3 minutes and 34 seconds credit?

`student`: I'll be responsible for maintaining my remaining time credit (he adds this to his CRH card), so I can answer that. Yes, I have enough credit.

`jukeBox`: `student`, have you played fewer than two tracks on this date?

`student`: Yes, I have not played two tracks today.

`jukeBox`: Okay, now we can play the track. Here it is, `cdPlayer`.

`cdPlayer`: Okay, `jukeBox`, I would be willing to tell the physical CD player to play this track. Actually, I have no idea how I am going to do that, but I'll write a `playTrack` responsibility on my CRH card for now as long as you send me the track to be played during a `playTrack` message.

This scenario has now reached a logical conclusion. However, the team feels that `jukeBox` already has enough responsibilities. "Why should I have to figure out if the student can play the selected track?" asks the member role playing `jukeBox`.

Chelsea suggests that they follow their instincts and try to distribute system intelligence as evenly as possible—a principle of good object-oriented design. Another heuristic for good design is avoid all-powerful classes. This suggests that in a good design, `jukeBox` might not be the best class for determining whether or not a student can play a track. Perhaps some other class should have that responsibility. The person playing `student` thinks it is appropriate to let the `student` object be responsible for figuring out if its human equivalent is allowed to play the selected track.

12.2.2.4 Alternative 2

`jukeBox`: `student`, can you play this track?

`student`: I feel as though I should be responsible for maintaining my own time credit. It seems appropriate that I should also know how many tracks I've played today. So I should be able to do some simple calculations to give you the answer you seek. Yes, `jukeBox`, I can play the track you sent me. I'll add these responsibilities to my CRH card:

- know how much time credit I have left
- know how many tracks I've played on this date
- respond to a message like `student::canSelect(currentTrack)`

The team wonders which alternative is better. One way to assess the design is to ask what feels better. Alternative 2 feels better somehow. Wouldn't it be nice if there were some object-oriented design heuristics to make us feel better about feeling better?

Well, it turns out that the first alternative has a higher degree of *coupling,* which means more messages are sent from `jukeBox` to `student`. There were two different messages versus the single message of the second alternative (`canSelect`). Chelsea adds this heuristic to the repertoire:

Object-Oriented Design Heuristic 12.5

Minimize the number of messages between a class and its helper.

Additionally, the second alternative has better cohesion. This means that the two knowledge responsibilities necessary to answer a `canSelect` message are closely related.

The Responsibilities of Each `student`

1. know how much time credit I have
2. know how many tracks I've played on this date

Additionally, the first alternative requires `jukeBox` to know about the internal state of the track and the internal state of `student`. This is a violation of Object-Oriented Design Heuristic 6.1, "All data should be hidden within its class." In conclusion, the second alternative delegates responsibility to the more appropriate class.

Chelsea (the object expert) verifies that the team member holding the `student` card should add `canSelect` as a responsibility. The team feels this scenario has reached its logical conclusion.

Self-Check

12-5 Summarize the algorithm that lets one user make one selection. You may send any message that you desire to any object you desire. Use any of the objects shown next as if the classes were already implemented. Add any message you like.

You are currently designing. Pretend you will be passing off your CRH cards and the entire algorithm to another programmer who will have to make it all work according to your design.

```
student currentStudent;
track currentSelection;
cardReader myCardReader;
cdCollection myCdCollection;
studentCollection myStudentCollection;
trackSelector myTrackSelector;
cdPlayer myCdPlayer;
```

There are many other possible scenarios. For instance, what if the student does not exist in `studentCollection`? What should happen? Or what if a student does not have enough credit or has already selected two tracks?

12.2.2.5 Scenario 2: "What happens when a student who has already played two tracks on this date tries to select a third track?"

`jukeBox`: Let us skip up to the point where I send a `canSelect` message.

`student`: No, I can't select a track.

`jukeBox`: I could simply send an appropriate message to the user.

So far, so good. However, Steve, who is holding the `cdPlayer` card, wonders when and how a track ever actually gets played by the physical CD player so it can be enjoyed over the stereo system. It seems as if `cdPlayer` needs some stimulus. The team has questions. Which class is responsible for sending the message that plays a track? And what happens when another track is already playing? If there are many tracks to be played, when and how will more than one track be played? Will the CD player be able to tell anyone when a track has finished? Misty asks, "Who is role playing the play list?" The team members check the lists of classes and to their dismay, discover that there is no `playList` class.

Chelsea comforts the team. "This often happens. We are trying to discover classes that represent an abstract view of the system. It is not unusual to discover useful abstractions at any point of software development. So feel free to add a `playList` class now." `playList` should maintain the list of tracks selected by users.

12.2.2.6 `playList`, a new class to maintain track selections in order

Charlie wants to know if `playList` should be a data member of the `cdPlayer` class? Don't CD players maintain their own play lists? Yes they do. Steve, who is role playing `cdPlayer` suggests that he contain `playList` as a data member.

The object expert suggests that this decision could be made at a later meeting. However, the team wants to force the issue now, as it is difficult to arrange a meeting time that all members can attend.

Jessica and Jason then lead a conversation suggesting that the overall system would be easier to implement if the physical CD player maintained `playList`. First of all, there is reuse of existing software (play lists) and hardware (CD selection mechanism, a clock, and hardware to read digital songs and convert them to real sound). Modern CD players not only maintain their own play lists, they also know precisely when to play the next track! If the system under development has to send `playTrack` messages at the appropriate time, `jukeBox` or somebody else would also have to maintain the time remaining on the currently played track—independent of the physical CD player. And how long does it take to load up CD #199 after CD #3 has just been played—3 seconds, 10 seconds, 5 seconds, 1 minute? When should `jukeBox` ask `cdPlayer` to play the next track? Will a track be cut short? Will there be unnecessary delays?

Chelsea begins a discourse that addresses a recurring design decision with systems that have concurrent (simultaneous) processes. Since the primary responsibility of `jukeBox` is to coordinate activities, `jukeBox` could control when the next track will be played. This could happen by using what is known as a*polling design*. `jukeBox` would go out and frequently poll (ask) `cdPlayer` if it is playing a CD. If `cdPlayer` replies "Yes," `jukeBox` can go off and do other things. If `cdPlayer` replies "No, I am not playing a track," `jukeBox` could ask `playList` for the next track and then send a message to `cdPlayer` to play it. In this case `jukeBox` continuously polls `cdPlayer`. In fact, this sounds a lot like `waitForUser`, a responsibility accomplished by polling—continuously asking `cardReader` if a card has been swiped.

It is also possible to have an interrupt-driven design. Each time `cdPlayer` finishes playing a track, `cdPlayer` asks `jukeBox` for the next track to play. In this case, `cdPlayer` is said to interrupt `jukeBox`. The `jukeBox` accommodates by getting the next track as soon as possible.

Matt reminds the team they have two computers. One is the PC that interacts with users as it maintains `studentCollection` and `cdCollection`. The second is a processor inside the physical CD player. Why can't we reuse the existing software and hardware of the CD player? Matt goes on to tell the team that this is easily done by setting the CD player to its internal program mode. His CD player at home can queue up to 99 different tracks at any one time. Charlie says he could deal with either a polled design, an interrupt-driven design, or this third option of having concurrent processes perform their responsibilities simultaneously on two separate processors.

Chelsea repeats, "Now that would be reuse!" She suggests that the team role play a scenario assuming `playList` resides in the physical CD player and follow a concurrent process design. And even though `playList` is already implemented in the physical CD player with a separate microprocessor, it might be useful to keep `playList` as a class during role playing. "It really feels important," suggests Blaine.

Chelsea asks the team to simplify the analysis. "All we need to do is send a `playTrack` message to `cdPlayer`. The `cdPlayer` class will be responsible for communicating with the physical CD player." The CD player will continuously play songs in a first-to-last order all by itself while `jukeBox` is busy getting new track selections from users. In fact, if the computer fails, the physical CD player could play out the entire `playList`.

12.2.2.7 Scenario 3: "What happens when a student is willing and able to play a track, but several tracks are waiting to be played?"

Chelsea tells `trackSelector` to pick up a scenario that already has a valid student.

`jukeBox`: `trackSelector`, here is `cdCollection`. Please return the user's musical selection.

`trackSelector`: Okay, I now have the `cdCollection`, so user, please select a track. The user selects the "I'm the Cat" track from Jackson Browne's CD titled *Looking East*. `trackSelector` returns this selected track, which `jukeBox` knows as `currentSelection`—an instance of the `track` class.

`jukeBox`: `student`, can you play the `currentSelection`?

`student`: Yes.

`jukeBox`: `cdPlayer`, please `playTrack(currentSelection)`.

`cdPlayer`: Okay, I got it. However, a track is currently playing and there are other tracks to play before I can play your selection. What do I do now?

`playList`: Hey, that's my job. I'll add the `currentSelection` at the end of the queue (waiting line) of tracks to be played, if that's what everyone thinks is fair.

Chelsea asks Blaine, the customer and domain expert, if this is the proper policy—first come, first served. Blaine retorts, "Yes, absolutely." Charlie, who has a part-time job as a network administrator at Lucent Technologies, pronounces that such a fair policy is easily enforced as a first-in-first-out (FIFO) waiting line—what computer scientists call a queue. Blaine asks Charlie how one can set up a waiting line inside a computer. Charlie says a queue is like a `vector` except that new things can be added only at the end. Things can only be removed from the front. Blaine reminds Charlie that he knows a lot about student policies but very little about `vectors` and queues. Chelsea reminds Charlie to avoid discussing implementation details during analysis—even if he knows about an existing `queue` class that can easily "queue up" tracks.

`playList`: Okay, I'll write these responsibilities on my CRH card: `queueUpTrack` and `getNextTrackToPlay`, even if I already exist in the CD player.

12.2.2.8 Scenario 4: "What happens when a student ID card is not found in `studentCollection`?"

`jukeBox`: Okay, I'm waiting for a user. Waiting, waiting, waiting . . .

`cardReader`: Okay, `jukeBox`, a user just swiped a card. Here is the student ID.

`jukeBox`: Thanks, `cardReader`. Now that I have the student ID, I need to verify if the user is valid. Hey, you know what? I am going to give a name to what I am doing. It seems like I am performing an operation that gets the student ID, finds the student, asks `trackSelector` for the student's selection, and so on. I'm going to summarize this algorithm and write it on my card as `processOneUser`.

`studentCollection`: Well, since it is my responsibility to store the collection of `student` objects, let me check to see if there is a student here with that ID. No, we have no student with that ID. What do we do now?

`jukeBox`: I'll just tell the alleged user that it's a no-go.

"Whoooooaaa," groans Jason. "A student with a valid student ID is a valid student. Let 'em in." Jessica supports Jason's suggestion by suggesting that "valid student" means something different each semester. Some students drop out in the middle of a term. New students come in at the beginning of a term. Jessica expresses concern about maintaining an accurate list of valid students—would we have to access the registrar's database of students? Blaine suggests that the cashless jukebox isn't a mission-critical system. Nor could anyone profit from its abuse. So even if a few "invalid" students get in, so what? The team members unanimously agree that a nonexistent student with a valid ID card should have an account created automatically. "And how much time credit do we give that new student account?" questions Matt. "1,500 minutes, of course," replies Misty.

`jukeBox`: Let me change my previous action so the new student is added to the new collection. Who should have that responsibility?

`studentCollection`: Why, me of course. Give me the student ID number from the magnetic ID card, and I will add a new student. I'll write the action responsibility `addStudent` on my CRH card. So `jukeBox` will not have to worry about the student returned. It may be someone who had been in the system, or it may be someone who was just added.

Self-Check

12-6 Play out the following scenarios by writing the class name and narrative of each until it has reached its logical conclusion:

-a What happens when a student swipes her card for the first time on a given date and wants to play two different songs?

-b What happens when a student has no more time credit left, but wants to play a song?

-c What happens when the Student Affairs office wants to remove a CD from the jukebox?

-d What happens when the Student Affairs office wants to add a new CD to the jukebox?

-e What happens when `cdPlayer` receives a `playSong` message and the physical CD player is turned off or malfunctioning?

12-7 List several college-library scenarios that should be played out by a team.

12-8 Script a college-library scenario that describes what happens when a user wants to check out a book.

12-9 Script a college-library scenario that describes what happens when a user wants to return a book that is not late.

12.2.3 Why CRH Cards?

After role playing, discussion, arguments, laughter, and changed minds, the design is captured as a set of classes that model a solution. Class names and responsibilities have been recorded on CRH cards as the team role played the scenarios. CRH cards help the problem-solving and system-building processes in many ways. Rebecca Wirfs-Brock writes [Wirfs-Brock 90]:

> We have found that index cards work well because they are compact, easy to manipulate, and easy to modify or discard. Because you didn't make them, they don't feel valuable to you. If the class turns out to be spurious, you can toss the card aside with few regrets. . . . If you discover you have erroneously discarded a class card, it is simple to retrieve it, or make a new one.

This is but one example of the many dynamics of CRH card use in software that go beyond the scope of this textbook. The previous jukebox discourse was an attempt to represent real-world object-oriented analysis and design.

In actuality, during October and November 1997, six students role played these and other jukebox scenarios. Some of the issues that arose are left as analysis and design exercises. For example, the team members strongly felt that no track should be played within the same hour. The author (acting as the customer and domain expert Blaine) agreed this would be a good enhancement, but it would be a change dealt with as an enhancement later. It is an interesting design decision to make. There are several possibilities.

One of the benefits of having a team consisting of designers and customers is the opportunity for everyone to develop an understanding of the system in common terminology. Sometimes customers haven't asked for what they really want. Experienced teams familiar with customer needs can actually help customers sharpen their requirements. The team can also make suggestions like this: "Please don't let the same track play over and over again. I couldn't stand that. It happened too much in high school. We had to turn the jukebox off." Limits of two tracks a day could prevent a lot of complaints.

12.2.4 Responsibilities and Helpers

The following summary of responsibilities and helpers is the result of the preceding CRH card development with the team role playing individual classes. These will be used in the next chapter to document the class names, action responsibilities, knowledge responsibilities, and collaboration for creating the class definitions. Knowledge responsibilities could become data members. Action responsibilities could become member functions. A helper might become a message send, a data member (containment relationship), an argument in a message, or a return value from a message. You will see more about these relationships and class definition design in Chapter 13.

FIGURE 12.7. *Major classes with responsibilities and helpers*

Class: `jukeBox`	**Helpers:**
Responsibilities:	`cdPlayer`
know current track	`trackSelector`
know current student	`student`
`waitForUser`	`cardReader`
`processOneUser`	`studentCollection`
	`cdCollection`

Class: `cardReader`	
Responsibilities:	**Helpers:**
`getStudentID`	physical card reader, which
	in turn collaborates with
	the magnetic student ID card

Class: `student`	
Responsibilities:	**Helpers:**
know remaining credit	`date`
know how many songs played today	
`canSelect`	

Class: `studentCollection`	
Responsibilities:	**Helpers:**
know all students	`student`
`getStudent`	
`addStudent`	

Class: `CD`

Responsibilities:	**Helpers:**
know tracks	
know CD title and artist name	
allow references to individual tracks	
know play time	

Class: `cdCollection`

Responsibilities:	**Helpers:**
know all CDs	`CD`
allow references to individual CDs	
`addCD`	
`removeCD`	

Class: `trackSelector`

Responsibilities:	**Helpers:**
`track getTrack(int)`	the user
	`track`
	`CD`
	`cdCollection`

Class: `track`

Responsibilities:	**Helpers:**
know play time	
know physical location in the CD player	
(CD number, track number)	

Class: `cdPlayer`

Responsibilities:	**Helpers:**
`playTrack`	the physical CD player
	`playList`
	`CD`

Class: `playList`

Responsibilities:	**Helpers:**
maintain list of selected tracks	`track`
`getNextTrackToPlay`	`queue`
`queueUpTrack`	

12.3 An Uninterrupted Scenario

Chelsea congratulates the team. The responsibilities seem to be distributed over a reasonable number of classes. If you consider that `cdPlayer` and `playList` are already implemented, the team has to consider and implement only eight classes at once.

Steve complains that he is still a bit confused. Each time a scenario began, it was not completed without some interruption. Chelsea had a lot to say about object-oriented analysis and design. Blaine sometimes had to address policy issues such as the play policy. Charlie just seemed to say too much about implementation. Additionally, it seemed that while the team members were trying to understand the problem, they also had to stop and make design decisions; should it be a polled design, an interrupt-driven design, or a concurrent design?

Jessica and Jason admit to some confusion. Jessica proposes that the team role play a scenario without interruption. This may be a repeat of what has been done, but it might help us understand the system better—a summary of sorts. Chelsea believes this is a good idea. She suggests the team hold the same CRH cards and role play a scenario.

12.3.1 Summary Scenario: "What happens when a student without an account wants to select a track?"

`jukebox`: I'll start the scenario again. I'm `waitingForUser`, so `cardReader`, please `getStudentID`.

`cardReader`: No one has swiped their card.

`jukeBox`: Okay, `cardReader`, I'll ask again. Please `getStudentID`.

`cardReader`: No one has swiped their card.

`jukeBox`: Okay, I'm waiting for a user. Waiting, waiting, waiting . . .

`cardReader`: Okay, `jukeBox`, a user just swiped a card. Here is the student ID: 1234.

`jukeBox`: `studentCollection`, please `getStudent` with ID 1234.

`studentCollection`: Well, there is no such student, so I'll create a new student account and the student will have a time credit of 1,500 minutes and no songs played on this date. I suspect this would be a default `student` object.

`jukeBox`: Thanks, `studentCollection`, for the `currentStudent`. `trackSelector`, I'm giving you `cdCollection`. Would you please give me the user's `currentSelection`?

`trackSelector`: Okay, I'll use `cdCollection` to show all the CDs and tracks to the user. The user will have to select the track. Got it.

`jukeBox`: Okay, I have the `currentSelection`. `student`, can you select a song?

`student`: Yes, I can select a song.

`jukeBox`: Now, `cdPlayer`, please `playTrack(currentSelection)`.

`cdPlayer`: `playList`, please add this track to the music we'll eventually play.

`playList`: Okay, I can queue it up, but if you aren't currently playing anything, why don't you play it now?

`cdPlayer`: Well, I am playing some good stuff now, so please queue it up.

`playList`: Okay, I'll add it.

`jukeBox`: Actually, I don't need to worry about what you guys did, I'm already waiting for another user.

Chelsea suggests that this scenario has reached its logical conclusion. `jukeBox` is waiting for another user. The CD player is playing the selected tracks in an FIFO order from `playList`.

Jessica, who has been role playing `cdCollection`, observes that she played no role in the scenarios. "You sent me to `trackSelector`, but I was never asked for anything. Why did `trackSelector` need me?" Chelsea agrees that something is amiss. It appears that `cdCollection` must make individual CDs available to `trackSelector` in some reasonable order. "What about artist names alphabetically?" asks Jessica. Blaine replies, "Seems reasonable. It's probably easier for students to pick an artist first, then a CD by that artist, then a track from that CD."

Jessica recalls that `vectors` allow access to individual elements in the container of objects. So it seems reasonable that `cdCollection` must also allow access to individual CDs from first to last. Charlie says he can handle it. Don't worry about it. Chelsea thanks Charlie.

Now everything is wonderful, except . . . What happens if the same user wants to play a second and then a third track? A new account will not be created this time, but somehow `student` must know if it `canSelect`. This new account should not be able to select a third track. No student should have unlimited time credit. In other words, `student` must be modified to reflect the fact that it selected a track and the CD player played the music. When should student updates occur?

Chelsea, who was role playing `student`, suggests that she could have updated herself when asked the `canSelect` question. She had the `currentSelection`, so she could have deducted the time it takes to play that track during `canSelect`. She could also have recorded today's date. When a second track was selected, she could have modified herself again in the same way. When the third `canSelect` message was sent to `student`, `student` could simply reply, "No can do. And by the way, I won't deduct any time, nor will I record the fact that I selected a third song on today's date."

The team can live with this, except Matt wonders how `studentCollection` will know if one of its students has been modified. Charlie says `studentCollection` could remove the old student and add the modified student. Blaine doesn't like that suggestion. Charlie says, "Okay, simply return a reference to the student so any change to `student` by `jukeBox` will also update the `student` in the `studentCollection`." Blaine bemoans, "What's a reference?" Charlie stops before he says a word. Instead, he quietly contemplates how to change `students` and maintain them so they are the same later in the day and even later in the term or in a student's career. Charlie has completed two computer science courses and he knows that references and files will prove useful helper classes.

The team meeting is over after Chelsea sets up a meeting with Charlie and Matt. Stay tuned for the class definition design and member function implementations (see Chapter 13).

Chapter Summary

- Object-oriented software development begins by identifying the key abstractions—classes—that potentially model a solution. Software designers assign responsibilities to the appropriate classes.
- Analysis and design decisions can be documented as Component/Responsibility/Helper (CRH) cards. Each CRH card begins as a blank 3-by-5-inch index card (or it could be 4 by 6 inches) with a class name and major responsibilities written on the back.
- Collaborative design and role playing enhance the object-oriented development process. Possible analysis classes arise not only from the problem specification itself and domain expertise, but also from the words that are spoken as teams analyze problems.
- Team members assume the roles of these analysis classes, play out scenarios (what would happen when . . .), and establish relationships between classes. During role

playing, team members establish more detailed responsibilities—what an object should know and what it should be able to do—while recording them on the CRH cards.

Exercises

1. The students are complaining. The same song plays again and again. One person plays it twice and then has a friend play it twice. The Student Affairs office asks you to modify the jukebox so it will not play any song that has been played in the previous 60 minutes. The original team has graduated. You have to do it by yourself. List all classes that must be modified. What changes must be made to each?

2. What changes would need to be made to the jukebox design in order to maintain the CD collection in this way: The jukebox has been running for a while. The Student Affairs office wants to replace the 10 least frequently played CDs with a fresh 10. The original team has graduated. You have to do it by yourself. List all classes that must be modified. What changes must be made to each?

3. After the jukebox has been running for a while, students begin to complain because they can no longer select songs due to the fact that that they are out of time. The Student Affairs office asks you to modify the jukebox so users can play up to 3,000 minutes of music. The original team has graduated. You have to do it by yourself. List all classes that must be modified. What changes must be made to each?

4. After the jukebox has been running for a while more, students begin to complain because they can no longer select songs due to the fact that that they are running out of time again. The Student Affairs office asks you to modify the jukebox so there is no time restriction. The original team has graduated. You have to do it by yourself. List all classes that must be modified. What changes must be made to each?

5. Students want to be able to play entire CDs on the weekends. The Student Affairs office asks you to modify the jukebox so each student can play one entire CD on any given date. The original team has graduated. You have to do it by yourself. List all classes that must be modified. What changes must be made to each?

Analysis Tips

1. Model the real world when possible.

Meaningful well-named abstractions can make the system design easier to understand.

2. Anthropomorphize.

Don't be afraid to give human characteristics to your objects. Ask these questions often:

- What should an instance of this class be able to do?
- What should an instance of this class know?

3. On your CRH cards, write action responsibilities as if they were C++ identifiers (no spaces).

This will help as you move on to designing class definitions. Conversely, write knowledge responsibilities like this:

- know all CDs
- know all tracks
- know play time

4. Do not procrastinate.

Write down the responsibilities as soon as you realize them during role playing. Early bouts of laziness can end up in frustration later.

5. Distinguish objects that are outside of the system under development.

For example, the jukebox will communicate with the user. The jukebox will read the student ID card. However, both the user and the magnetic ID card are outside of the system.

6. The user is often confused as being a key abstraction to be modeled.

The user is important. However, there is usually some state object that models the real-world user. Remember that there is a physical user that selects songs, but there is also a software equivalent called `student` that knows how many songs the software equivalent has played today.

7. Definitely draw a picture of the major classes.

Make the classes rectangular boxes with the class name and the major responsibility (10 words or less). The picture should have arrows from the sender of a message to the receiver of that message (see the `jukeBox` picture).

8. Helpers usually represent a one-way relationship.

If class A asks class B for help, A is the sender, B is the helper. Write down that responsibility on B's card. `jukeBox` asked a helper named `studentCollection` to `getStudent`. `getStudent` should be on the `studentCollection` card.

9. It is not a good idea to ask the physical user if he or she can select.

However, it is perfectly okay to ask the software object. The physical student might say, "Sure I can play 100 songs today." Assume the programmer will not let the user's software equivalent lie. Besides, the user is not part of the system.

10. The first round of scenarios will often be interrupted by discussions.

Interruptions are okay. This is called brainstorming. Five heads are better than one. Genius is more easily accomplished with more than one person. At some point, have your team play out a familiar scenario without interruption. You will see the objects more clearly and understand their responsibilities.

11. The jukebox and the projects at the end of this chapter are relatively large systems.

They certainly do more than convert from Fahrenheit to Celsius. If you don't understand `jukeBox`, don't worry. The best way to understand it is to do an analysis with a team. This could take hours. On more complex systems, this could take months. The projects of this chapter are less complex. However, they are complex enough to benefit from an object-oriented approach to software development.

12. The jukebox has key abstractions that set a pattern for the analysis projects coming up at the end of this chapter.

As you perform your own analysis and CRH card development in a team project (12A, 12B, 12C, 12D) look for the following major classes (the analysis projects typically have five, six, or seven classes). Like the jukebox, they include

- a class that coordinates the major activities (`jukeBox` here)
- a collection of objects (`studentCollection` and `cdCollection` here)
- state objects (`student`, `track`, and `CD` here) stored in a container class
- one or more classes that model something in the real world (`student`, `cardReader`, and `cdPlayer`)

Object-Oriented Analysis/Design Projects

Each of the following analysis projects assumes you are using a team approach involving activities like those described in this chapter. Team size should be two, three, four, or five students.

1. Once you have a team, pick a project you want to do. Choose from this list:

   Bank Teller (12A)

   Voice Mail System (12B)

   Video Rental Store (12C)

   Checkbook (12D)

   make one up

2. Next, analyze the problem to establish a reasonable set of classes that model a solution. Draw the classes on a piece of paper. Make sure each class has a name and a major responsibility assigned to it.

3. For each useful class, write the name of the class and its major responsibility on the back of a 3-by-5-inch index card (or an 8½-by-11-inch piece of paper if you don't have index cards). On the front, list the class, the more finely detailed responsibilities, and any helpers the class uses to accomplish its task.

4. Run through scenarios that you invent. Using a pencil, write down responsibilities as they crop up. Remember, the class may have action responsibilities, knowledge responsibilities, or both. Also write down any helpers that exist. If you ask another object for help, that person (object) is a helper. You may erase, you may

cross off, or you may add responsibilities; in fact you may even tear up and recycle that CRH card.

5. Role play as many scenarios as you can think of. Do them until you really understand the system. The CRH cards should provide an accurate portrayal of the responsibilities of each class.

Save your CRH cards. They will be used to help you design class definitions (Chapter 13). Your project can be continued into the next chapter and on into complete implementation.

12A Bank Teller Application Adapted from *Problem Solving and Program Implementation* [Mercer 91]

First read the notes above. The bank teller application allows any bank customers access to their own bank accounts through their customer numbers. Once a customer swipes the bank card and enters the personal identification number (PIN), the user, with the help of a teller, may complete any of the following transactions: withdraw money, deposit money, and query account balance. Customers may also see their own transaction logs. The system must maintain the correct balances for all accounts and also log each and every successful transaction.

12B Voice Mail System Adapted from *Mastering Object-Oriented Design in C++* [Horstmann 96]

First read the notes above. Simulate a voice mail system. The system has a collection of mailboxes, each of which may be accessed by an extension number (3445, for instance). A user may put a message into any mailbox, so anyone on the computer may type in a mailbox number and then type in a message. Any user with a valid mailbox and the valid password may do any of the following:

- play back messages
- delete messages
- change the greeting
- change the password

An administrator is needed to activate new mailboxes and deactivate active mailboxes. The administrator is a user with a "super password."

12C Video Rental System from *Data Structures via C++* [Berman 97]

First read the notes above. Build a software system to support the operation of a video rental store. The system should automate the process of renting tapes and receiving returned tapes, including the calculation and printing of customer bills, which may or may not be done at the same time a tape is returned. The system must also give the clerk access to information about the tapes, such as the number of copies on the shelf of any given video owned by the store. The system must be able to add and remove customers and tapes to and from the database. Each customer and each copy of each tape are associated with a unique bar-coded label.

12D Checkbook Application Adapted from *Using CRC Cards* [Wilkinson 95]

First read the notes above. Simulate an electronic checkbook. The checkbook manages a limited set of entries: checks written and bank deposits. The checkbook must maintain the proper balance and a record of all entries. The checkbook will be able to print a statement of all activities. The user should be able to look up any check number individually and see the amount and who the check was written to.