Chapter Sixteen

Object-Oriented Software Development

Inheritance and Polymorphism

Summing Up

The previous chapter introduced dynamic memory allocation and a linked list. The notions of indirection and dynamic memory allocation, introduced in the preceding two chapters, will be used in this chapter to store a collection of dissimilar elements.

Coming Up

This chapter uses a team-based approach to introduce the two other major features of the object-oriented paradigm:

- inheritance: the ability to derive a new class from an existing class
- polymorphism: the ability of different types of objects to respond to the same message in different ways

A case study presents another object-oriented approach to software development. Along the way, the team discovers a class hierarchy that provides experience with inheritance, polymorphism, and heterogeneous collections. After studying this chapter, you will be able to

- recognize generalization that may be implemented with inheritance
- derive new classes from old ones
- override member functions and add new features to derived classes
- apply object-oriented design heuristics for inheritance
- understand and use polymorphism

16.1 Discovery of Inheritance through Generalization

This section provides another case study in object-oriented software development. This time the problem is from the domain of a college library system. It follows the same methodology presented with the Chapter 12 cashless jukebox case study. This problem is very similar to a problem described in Nancy Wilkinson's book *Using CRC Cards* [Wilkinson 95]. Related items such as a library class hierarchy, inheritance, polymorphism, a date class, and a data structure capable of storing different classes of objects are described in *Problem Solving and Program Implementation* [Mercer 91].

THE PROBLEM STATEMENT: *College library application*

The college library has requested a system that supports a small set of library operations: students borrowing items, returning borrowed items, and paying fees. Late fees and due dates have been established at the following rates:

	Late Fee	Borrowing Period
Book:	$0.50 per day	14 days
Videotape:	$5.00 one day late plus $1.50 for each additional day late	2 days
CD-ROM:	$2.50 per day	7 days

A student with more than seven borrowed items, any one late item, or late fees greater than $25.00 may not borrow anything new.

Object-oriented software development attempts to model a real-world system as a collection of interacting objects—each with its own set of responsibilities. This helps organize the system into workable pieces. The three-step object-oriented software development strategy introduced in Chapter 12 is repeated here for your convenience:

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 each object must know about itself (data members).

3. Determine the helper classes for each. To help complete its responsibility, a class typically delegates responsibility to one or more other objects. These are called helper classes.

The team consists of the library domain expert, Deena, and five students who are analyzing the college library system as part of a computer science honors option: Jessica, Jason, Steve, Matt, and Misty. Austen is the object expert this time.

16.1.1 Identify the Classes

The team now has some experience with object-oriented software development. They plan to use their experience. The first goal, or deliverable, is a set of potential classes that model the problem statement. Each class will describe its major responsibility. The team starts by writing down all the nouns and noun phrases in the problem statement, redundant entries not recorded.

Nouns (from the problem statement)

college library	system	library operations	CD-ROM
librarian	student	item	seven borrowed books
fee	book	videotape	
late fee	day	due date	

The following list represents the set of potential classes for modeling a solution:

Potential Classes: *A first pass at finding key abstractions*

Somewhat Sure	Not Sure
librarian	system
student	operations
book	late fees
video	due date
CD-ROM	day
seven borrowed books	fine
college library	

Matt recommends keeping `librarian` as the name for a class responsible for coordinating the activities of checking books in and out.

A `student` class will be a useful key abstraction. After all, students will be checking books out, checking them in, and paying fines. The domain expert, Deena, mentions that `librarian` should also be allowed to lend books to faculty, staff, and other members of the community. After all, it's a state university paid for in part by state taxes. So the team decides to change the class name to `borrower` to reflect the general notion of someone who can borrow a book from the library.

Misty doesn't believe "seven borrowed books" should be a key abstraction. One of the programmers on the team chimes in and suggests, "This is not a problem. I can see this as a `bag` of `book` objects." The domain expert Deena remarks, "A book bag?—cool." Since the team is currently looking for classes that help observers understand what the system does rather then how the system will eventually do it, the team decides that "seven borrowed books" is not a class. Instead, this is a knowledge responsibility.

Austen uses some object-speak: "A `borrower` should know its own collection of borrowed books." Matt jumps in and asks Austen, "Aren't CD-ROMs and videotapes in the same category? They can be borrowed too!" Deena agrees. Austen states that Matt has implicitly discovered a basic concept of object-oriented analysis. Some classes have things in common. In this case, there are several categories of things that can be borrowed: books, CD-ROMs, and videotapes. The team considers the responsibilities these classes have in common. After some discussion, Jason and Jessica produce the following list of responsibilities that all three classes of objects have in common. Each `book`, `cdRom`, and `video` should:

- know its due date
- compute its due date
- determine its late fee
- know its borrower
- check itself out
- check itself in

Self-Check

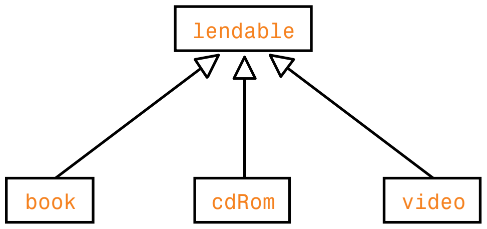
16-1 Which responsibilities are the same for these three classes?

16-2 Which responsibilities should be carried out differently?

Jason notices that the term "borrowable item" has been floating around. Austen suggests there could be an abstraction named `borrowableItem` that supplies the attributes and behavior common to all items that could be borrowed from the library. Although differences exist in the computation of late fees and the setting of due dates, each `borrowableItem` has several common responsibilities. Steve complains that `borrowableItem` is a bit of a tongue-twister. Matt suggests `lendable`: "`lendable` represents things that are, well, lendable." Deena likes this new name. `lendable` it is.

Austen points out that the team has intuitively discovered the inheritance relationship between classes. *Inheritance* is another name for generalization. Matt instinctively saw several seemingly different objects and found some common things. He generalized. Since no one has seen or heard of the inheritance relationship, Austen draws the following diagram on the chalkboard:

Figure 16.1. *The inheritance relationship in UML notation (`lendable` is the abstract class)*



Austen explains that the C++ culture would refer to `lendable` as the *base class*. The other three classes—`book`, `cdRom`, and `video`—are known as *derived classes*. The member functions and data members common to all lendables should be listed in the base class (`lendable`). Through inheritance, each derived class (`book`, `cdRom`, and `video`) inherits member functions from the base class (`lendable`). Austen then displays a design heuristic in order to explicitly encourage the team to accept the inheritance relationship between classes as good design.

Object-Oriented Design Heuristic 16.1 (Riel's 5.10)

If two or more classes have common data and behavior, then those classes should inherit from a common base class that captures those data and methods.

The common data and operations include:

- a date for knowing the borrower and the due date
- operations such as check self out and check self in

However, without some difference amongst the derived classes, there is no reason to use the inheritance relationship. Instead, there should be just one class. So, in addition to commonalities as mentioned above, there must be enough differences to justify having more than one class. There are two, possibly three, major differences between the three classes:

1. computation of the due dates
2. computation of late fees
3. different attributes (`video` has a movie-studio attribute, for example)

Jason is confused and questions Austen. Jason doesn't understand why there is a `lendable` class in the library system. Austen decides it might prove useful if he explained the difference between an abstract class and a concrete class. An *abstract class* describes the data and operations meant to be common to all derived classes. An abstract class cannot be instantiated. Therefore, this code should be rendered illegal:

```
lendable aLendable;  // ERROR--attempt to construct abstract class
```

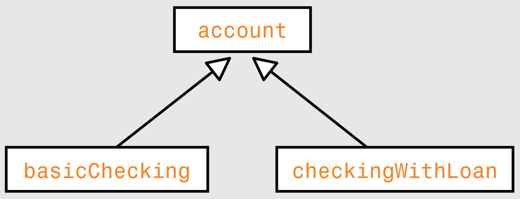
Abstract classes exist to capture common operations and data. They are not to be constructed.

A *concrete class* is one that can be instantiated. Therefore, these constructions must be legal:

```
book aBook;  // To be implemented later
cdRom aCDROM;
video aVideo;
```

Self-Check

16-3 List the abstract class in the following `account` hierarchy:



16-4 List the concrete class(es) in this inheritance hierarchy.

16-5 List an operation that would make one class different from another.

16-6 List one operation that would be the same for the concrete classes.

16-7 List one data member that both derived classes would likely have.

16-8 List one data member that would exist in one class, but not in the other.

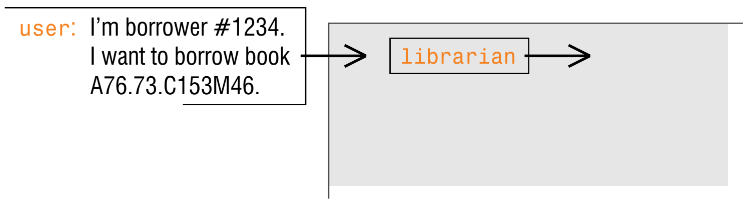
Jessica claims to understand the notion of the inheritance relationship, although the implementation is not in her head yet. Austen remarks, "I could show you how all this works, but don't worry about it for now. I'll show the implementation details in a bit." Matt articulates that the team should get back on task with analyzing the college library system.

Jason reminds the group that the following classes have been identified so far:

- `librarian`
- `borrower`
- `lendable` (and the derived classes: `book`, `cdRom`, and `video`)

With some consensus realized, Steve wants to know what the user interface will look like. Will there be text-based input and output communication between the user and the system? Is a graphical user interface desired? What about a card reader or touchscreen for input? Steve enjoyed the touchscreen ordering system recently tested at the local Taco Bell™. Misty points out that it has since been removed. Deena, the domain expert, isn't sure what the interface should look like. She believes the library will be okay with a text-based interface to the system. Text-based input and output are acceptable. Matt draws a picture so everyone can understand the relationship between a user and `librarian`:

Figure 16.2



Steve wonders whether `user` should be included as one of the classes to model the problem. Misty reminds Steve that the user is a person who approaches `librarian`, shows identification, and makes a request. `borrower` represents an object inside the system that knows everything about that user that the system will need to know. For example, `borrower` could also be responsible for knowing if its human counterpart (the user) owes any late fees. User is not going to be a class. However, it could play a part in the scenarios. Austen confirms that this is a good decision. Matt lets Austen in on a little secret: "We went through this once before with the cashless jukebox that's jammin' in the student center."

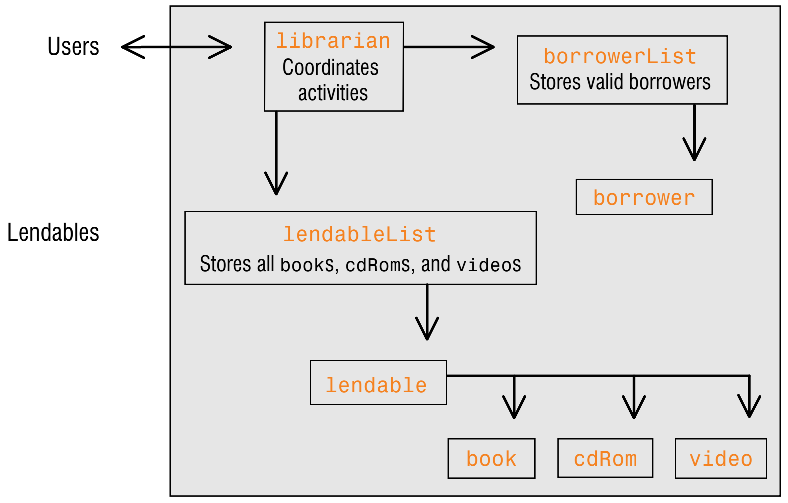
Deena has a problem with the concept of a `book` class and a `borrower` class. Certainly the system must maintain many borrowers and many books. Matt relates, "One class can create many instances, or objects. Since there will be many lendables and many borrowers, we should probably add two new classes named `borrowerList` and `lendableList` to store, retrieve, and delete `borrowers` and `lendables`, respectively." The team agrees. It worked for the jukebox.

The team feels as though they have captured several key abstractions (classes) for this application. They now have a framework for analyzing the problem in more detail. There is a sense that the primary responsibility of each class has been recognized. The team documents their progress with a table that lists class names along with their primary responsibilities.

Class Name	Primary Responsibility
`librarian`	Represent object responsible for coordinating activities of checking books in and out.
`borrower`	Represent one instance of someone who can borrow a `lendable`. There may be thousands of `borrowers`.
`lendable`	Represent an abstract class from which many "borrowable" items can be derived. This abstract class captures the common member functions and data members of any item that can be borrowed from the college library.
`book`	Represent one book that can be checked in and checked out. There may be thousands of `books`.
`cdRom`	Represent one CD-ROM that can be checked in and checked out. There may be thousands of `cdRoms` stored in the database.
`video`	Represent one videotape that can be checked in and checked out. There may be thousands of `videos`.
`borrowerList`	Retrieve, delete, add, or update any `borrower` from the thousands of `borrowers` in the database of valid `borrowers`.
`lendableList`	Retrieve, delete, add, or update any `lendable` from thousands of "borrowable" items.

This following picture provides an abstract view of the system so far. It also marks the boundaries of the system. Everything in gray is in the system under development. The users and physical items that can be borrowed are outside the system.

Figure 16.3



16.2 Refinement of Responsibilities

With primary responsibilities identified, the team now sets about the task of identifying and refining other responsibilities. The team will also identify helpers—the other classes needed to carry out a specific responsibility. The team will try to answer questions such as: What are the responsibilities? Which class should take on each of the responsibilities? and What class(es) help accomplish another class's responsibility? Responsibilities convey the purpose of a class and its role in the system. Analysis is enhanced when the team thinks that each instance of a class will be responsible for two things:

1. the knowledge each object of the class must maintain
2. the actions each object of the class can perform

Austen recommends that the team to be prepared to ask the following two questions:

1. What should an object of the class know about itself (knowledge)?
2. What should an object of the class be able to do (actions)?

Assigning a responsibility to a class means that every instance of that class will have that responsibility. This is true when there are many instances of the class—as in `lendables` and `borrowers`. It is also true when there is only one instance of the class—such as `librarian` and `lendableList`.

The next team activity involves assigning more specific responsibilities to the classes already identified. One technique involves role playing. Each team member assumes the role of one of the classes. The team members play out scenarios while adding responsibilities and helpers to CRH cards.

16.2.1 A "User Cannot Borrow" Scenario

At this point, team members assume the roles of the classes. They plan to try scenarios to see how one instance of a class interacts with other objects. A scenario is the answer to the question What happens when . . . ? The team decides the first scenario will be the response to this question:

16.2.1.1 Scenario 1: "What happens when user #1234 wants to check out a book and currently has seven borrowed `lendables`?"

`librarian`: Well, I'm the librarian so I guess I'll start. I just got a user ID. It's #1234. Now User, what do you want to do?

User: I want to check out a book.

`librarian`: Now I need to know the call number of the book.

User: The book's call number is QA76.1.

`librarian`: Okay, now let me verify that this user can borrow. I'll ask `borrowerList` to look up the `borrower` with ID #1234.

`borrowerList`: I found the `borrower` you asked for. I'm sending it back to you. I'll add `getBorrower` to my responsibility list. I must know all borrowers.

`librarian`: Thanks, `borrowerList`. Now I have the software object that represents the human user. I believe my job will be easier if I can send `borrower` a `canBorrow` message. I am now helping `borrower`. What about it, current `borrower`, can you borrow a new book?

`borrower`: Since I am responsible for knowing my borrowed `lendables`, I should be able to tell you if I have seven or more things checked out. Hey wait a minute, you didn't ask me if I had seven borrowed items. You asked me if I could borrow. So I am going to add a `canBorrow` responsibility to my list. I will return a code indicating that I can borrow. Will false or true do? To borrow, my late fine must be $25.00 or less and I must have fewer than seven borrowed items, none of which can be overdue. It seems like I'll need to know my own borrowed `lendables`. I have seven borrowed items. No, I cannot borrow.

`librarian`: Thanks, `borrower`. Things are made easier for me because I can delegate the `canBorrow` responsibility to you. Also, remember when we were working on the jukebox. Chelsea told us that we should try to distribute system intelligence as evenly as possible. We had a `canSelect` message to simplify the jukebox. Now, I can just send a `canBorrow` message to you. I think I now must send an appropriate message to the user that borrowing a book is not an option.

The team decides they have run this particular scenario to its logical conclusion. They also feel that there are obviously many possible scenarios to role play such as successfully checking out a book and returning a lendable. The team also wonders if users should be able to look up a book by call number to see if it is in or out. Perhaps a user might want to know when a book is due back in the library. This suggests that a `lendable` should know when it is due and it should be able to tell someone the actual due date.

Deena confirms that this is certainly desirable behavior. However, the problem specification does not list these requirements. The team agrees to plan for the possibility of adding such enhancements later.

16.2.2 A Check-Out Scenario

16.2.2.1 Scenario 2: "What happens when user #1234 wants to borrow a `lendable` with call number QA76.2, has three books out, none of which are late, and has late fines of only $5.00?"

`librarian`: I'll start again; I'll get the user ID.

User: My ID is #1234.

`librarian`: Let me ask `borrowerList` for the proper borrower. Please `getBorrower` with ID #1234.

`borrowerList`: I found the `borrower` you asked for. I'm sending it back to you.

`librarian`: User, what do you want to do?

User: I want to borrow something with call number QA76.2.

`librarian`: Now I have the current `borrower` and the current `lendable` in my possession. I think I'll check with `borrower`: can you borrow?

borrower: Let me see if I have fewer than seven borrowed items. Yes, I currently am borrowing three lendables. Now, are any lendables overdue? Let me ask the first lendable in my list of borrowed items: lendable_0, are you overdue?

lendable_0: I am responsible for knowing if I am overdue. I'll have to ask date to compare my due date with today's date. I will add the responsibilities "know due date" and isOverdue. I asked date for help.

"Who's date?" asks Deena. She wants to know what a date class would be responsible for. The team decides to add date as a key abstraction and see what its responsibilities are. One of the programmers writes down date as a helper on a CRH card and agrees to play the role of date.

date: To compare the due date with today's date, I must be able to get today's date and to compare two dates. Yes lendable_0, today's date is less than or equal to the due date. I'll add <= and todaysDate to my responsibilities.

lendable_0: Thanks, date. I am not overdue.

borrower: I'll check my other lendables. lendable_1, lendable_2? None are overdue. Since I also have no late fees, I can tell librarian, "Yes, I can borrow something new."

librarian: lendableList, please get me the lendable QA76.2.

lendableList: Okay, here is the lendable.

librarian: What should I do now?

The team pauses and considers a couple of possibilities. It seems as if the current borrower and the current lendable both need to be updated somehow to record that the lendable has been checked out. It seems logical to update the lendable first and then send it to the borrower to add to its list of lendables. It also seems appropriate to update lendableList and borrowerList. That ensures that all borrowers and lendables are accurately updated.

Austen recommends that the team first ask the question, What should be done to update lendable? The team decides that the book's status should become "not available." Also, lendable's dueDate should be set to the appropriate day in the future so later on the borrower can ask the book if it is overdue. The team also believes that it is important to know who has borrowed the lendable—in case someone wants to find out who has it. The team comes up with two alternatives.

The first alternative places the responsibility of updating a `lendable` upon `librarian`. The second alternative delegates this responsibility to the `lendable` itself. The team decides to role play both alternatives. Here is the first.

16.2.2.2 Alternative 1 (updating the `lendable`)

`librarian`: `lendable`, compute and set your due date.

`lendable`: Okay, I'll `setDueDate` to either 2, 7, or 14 days from today—depending on what class of `lendable` is currently being checked out.

Austen breaks in saying, "That's polymorphism!" The class can be determined while the program is running. The particular version of `setDueDate` will depend on the class of the object. And the class of object cannot be determined until the moment the `lendable` is being checked out—at runtime.

`lendable`: I'll set the due date. I'll add a `computeDueDate` responsibility to my CRH card.

`librarian`: `lendable`, now please record #1234 as your borrower.

`lendable`: Okay, I'll set my borrower ID as user #1234.

`librarian`: Okay, now mark yourself as not available.

`lendable`: Done.

16.2.2.3 Alternative 2 (updating the `lendable`)

`librarian`: `lendable`, you could be responsible for checking yourself out. If I tell you who the `borrower` is, could you check yourself out?

`lendable`: Sure. I add these new responsibilities to my CRH card: `computeDueDate` and `checkSelfOut`.

Which alternative is better? One way to assess the design is to ask yourself what feels better. Alternative 2 feels better somehow. But wouldn't it be nice to have some design heuristics to make us feel better about feeling better? Well, it turns out that the first alternative has a higher degree of coupling. There were three different message sends versus the single message send of the second alternative. Additionally, the second alternative has better cohesion—the three responsibilities of `lendable` accomplished by a `lendable::checkSelfOut` message are closely related.

1. Know my `borrower`.
2. Compute my due date and set my due date.
3. Update my availability status.

Additionally, the first alternative requires `librarian` to know more than is necessary about the internal state of the `lendable`. Alternative 2 delegates responsibility to the more appropriate class. So the team member holding the `lendable` card adds `lendable::checkSelfOut(borrower)` to the set of responsibilities on the CRH card.

Now, has this scenario reached its logical conclusion? No. The `borrower` does not know about its new borrowed `book`. Remember that it is the responsibility of each `borrower` to know its borrowed `books`. The `borrower` must be updated. Here is one conclusion to this scenario.

`librarian`: `borrower`, let's follow the design heuristic we just talked about. I'll just send this message: `borrower.checkOut(lendable)`

`borrower`: It seems as though I should be able to add a `lendable` to my list of borrowed `lendables`. I'll add `borrower::checkOut(lendable)` as a responsibility on my CRH card.

`librarian`: Please inform the user that everything is okay—the user may take the `lendable` along. Whoops, I almost forgot. I better update `borrowerList` and `lendableList`. `borrowerList`, please put this `borrower` away.

`borrowerList`: Okay, I'll add `putBorrower(borrower)` to my CRH card.

`librarian`: `lendableList`, please but this `lendable` away.

`lendableList`: Okay, I'll add `lendableList::putLendable(lendable)` to my CRH card.

`librarian`: So User, anything else?

User: No, I'm outta here.

`librarian`: Okay, I'm ready to process another user.

The check-out scenario has reached a logical conclusion.

Self-Check

16-9 Write a check-out algorithm that sends any message 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.