html,v 1.1 2003/06/11 19:39:26 jon Exp $

Introduction to Interactive Programming
By Lynn Andrea Stein
A Rethinking CS101 Project

Chapter 2.
The Software Development Process

Chapter Overview

- What does a software developer do?
- How does the programming process work?

This chapter builds on the previous one by walking through an example of interactive program design. It reiterates the central questions a programmer asks: What is the overall behavior of the program? Who are the members of the community? What goes inside each one? How do they interact? In addition, it stresses the ideas of incremental construction and testing (start with simple functionality and add on only when the program is working); repeated cycling through designing, building, testing, and back again; and the necessity of modifying and maintaining software on an ongoing basis. The example in this chapter is presented in English rather than in actual Java code; it is intended to introduce students to the idea of programming, to the processes involved, and to the kinds of questions that they will be asking throughout this book.

This chapter differs from the remainder of this book. The rest of the book concerns what you need to know to write programs. This chapter is about the actual experience of doing software development. It provides a context of use for the rest of the book. After all, there is no better way to learn to develop software than to do it, and programming requires a lot of practice. In the remainder of the book, you will learn the things you need to know to get the computer to perform certain tasks. In this chapter, you will learn how to work with the computer to apply that knowledge. Most importantly, this chapter describes the experiences that you will have in working on the programming laboratories that should accompany your use of this book.

Objectives of the Chapter

1. To understand the development cycle, its stages, and their interactions
2. To increase ability to recognize and articulate use cases from a problem description
3. To increase ability to recognize and articulate problem requirements including needs, constraints, and resources of the user, physical environment, and software environment
4. To increase ability to articulate appropriate assumptions and guarantees inherent in a design
5. To be able to use an engineering notebook to track the development process
6. To decompose problem requirements to identify the components (entities, objects, and actions) of a software solution
7. To recursively decompose these components (entities, objects, and actions) until individual recipes are reached
8. To be able to test a design by acting it out
9. To be able to write a staged development plan for constructing a piece of designed software
10. To design tests for various stages of this development plan
11. To understand the process of editing and compiling source code and running the compiled code; to understand when recompilation is necessary
12. From tests, to identify the presence of bugs and to propose strategies to locate and resolve them
13. To use printing statements, debuggers, and interpersonal interaction to locate and resolve bugs

2.1 The Development Cycle

The previous chapter explored what programs are made of. In this chapter, we'll look at the process by which programs are created and what happens to them as they continue to grow and change. Software development -- creating, modifying, and maintaining computer programs -- is often the job of a software engineer.

2.1.1 Software Development

In the previous chapter, we used six questions to think about program design:

- What is the behavior of this program?

If it is a community of entities, we need to figure out how it is put together; we need to decompose it:

- Who are the members of the community, the entities that combine to produce this behavior?
- How do these community members interact?
- What goes inside each one? What is each one made of? (A community of entities or a single instruction-following control loop?)

And, for each instruction-follower, we need to write its recipe:

- What does it do next?
- How does it do each one of these things?

As you design your program, answering these six questions, you will likely find that later decisions involve going back and modifying earlier parts of the design, changing them or specifying them in greater detail. You will probably also discuss your design with other programmers -- or, perhaps more importantly, to the users or customers for whom you are creating this service -- and revised your design specification in response to their feedback. As you have answers to these design questions, you can start to build your program (or at least a simplified version of it).

The implementation phase of the project is similar. In building a program that is supposed to meet your specification, you will often find that you need to go back and change (or at least add details to) that specification. When this happens, you need to be careful to consider all of the interdependencies that led you to your original design. That is, the development of software is cyclic, beginning with design but often returning to it. It will not always be desirable (or even possible) to change your design, but it is quite common to discover additional assumptions or nuances that must be percolated through the design during later phases of development.

When you begin to build your program, it is advisable to implement only a small piece of your system first. This may mean implementing only some of the entities, or it may mean implementing all of the entities but only simple, basic versions of each. In large scale system development, this initial phase is called prototyping. For example, you may build a restaurant in which there is only one thing to be ordered. [[Footnote: No coke, pepsi.]] Building a simple version first lets you see that you have gotten the basic structure right. As you get this version working, you can begin to add more complicated features -- such as varying what is ordered, making sure that the waiter can handle a variety of different requests -- one by one.

Even in most of the smaller scale programs that you will encounter in your early course work, it is a good idea to utilize this approach of incremental program development. Part of developing good programming skills involves learning to consciously and explicitly design a staged development plan in which smaller simpler programs are constructed and debugged, then gradually expanded until the desired functionality is obtained.

Building a simpler version of your system gives you an opportunity to test your basic approach before you have built up too much complexity. It also means that your ***bugs***, or program errors, will be easier to find. Bugs come in many flavors, ranging from simple syntactic errors such as spelling mistakes, to programming errors such as incorrect variable scoping, to conceptual design problems such as impossible-to-meet but critical guarantees.

Even after you've found the bugs that keep your program from running, you will need to subject your code to rigorous testing. This means trying out not only the "normal" expected behavior, but also checking how your program handles unexpected or anomalous behavior. Think of your program as an opponent you're trying to trick; see if you can get it to misbehave. This testing -- when done right -- will lead you to modify your code or even your design.

2.1.2 Software Lifecycle

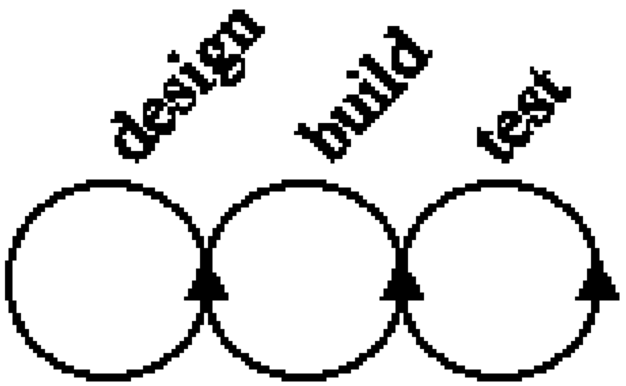

Figure 2.1. Software development is a continually cycling proces

How, then, does a programmer provide for this behavior? Software development is an intertwined process of designing, building, and testing. Each of these elements provides feedback to earlier phases of the development process. During the lifetime of a piece of software, the requirements that first shaped it will change and as they do so, the design and implementation of that software will need to change, too.

For example, the SmallTown library may decide to automate its catalog and circulation system. The new system should keep track of what books the library has by author and title as well as which books are checked out to which patrons. How does this software come into being?

Some people think about programming as though the goal were to produce a working piece of ***software***. They will describe what a programmer does as a step by step recipe intended to create this result.

1. Get requirements from user(s).
2. Design solution.
3. Build solution.
4. Test solution.

...producing a finished program.

This description of the programming process should sound reminiscent of the peanut butter and jelly model of program behavior. Just as making sandwiches -- producing results -- is an important part of program behavior, producing programs is an important part of a software developer's job. But a software developer is more like a restaurateur than a peanut butter and jelly producer, and producing a program is not the whole job. In fact, software development itself is an ongoing, interactive process.

This list of steps in the process of software development is sort-of right. The first part of building software is understanding the requirements that software will need to meet. A software developer producing the SmallTown library system would need to understand the properties of books and library patrons that should be tracked, the kinds of access librarians and the public need to the system, the types of reports on circulation that library administrators want, etc. These are the use cases of a library system, and use cases are always a good place to start.

But the list of steps makes it look like each of these pieces of software development -- understanding requirements, designing, building, and testing -- happens on its own, in sequence. In fact, the different pieces happen in an ongoing, overlapping, interrelated way. For example, after sketching a preliminary design for the library system, the software engineer might bring this proposal back to the SmallTown library administrators to see how well it fits their requirements, even before beginning to build the system. The software construction phase might begin by building a very simple system that allows a library staff member to enter book or patron data. Discussion of this system -- among designers, developers, users, and management -- might lead to a redesign in which different pieces of data could be entered and edited by different staffers at different times.

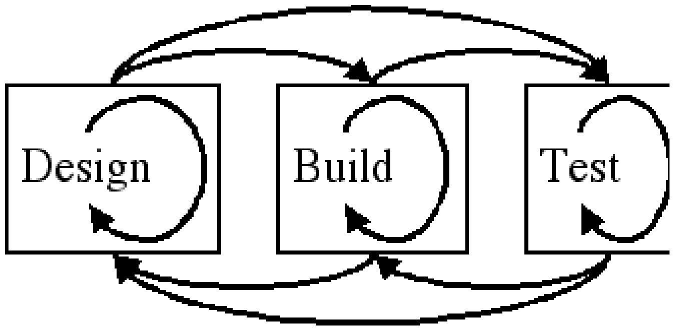

Figure 2.2. Design, build, test, always with feedback to earlier phases; each is an ongoing process.

As the initial prototype is built, each component and phase will need to be tested. Some of these tests will result in additional building; a few of them may even send the software engineers back to SmallTown to discuss requirements and design further. When the design seems settled, the software engineer can build a production version, but even then customer feedback -- and changing library needs -- may modify the software as it is built and tested. As the program is under construction, the library staff may come up with new benefits they'd like to see from the software, or the software engineer may be able to provide the library with additional flexibility, by letting building and testing influence design.

This scenario -- modification of an existing piece of software -- is actually the norm. Development of new software from scratch is the exceptional case. But even in that software developer use case -- the one for which the peanut butter and jelly recipe seems to be an answer -- there is more interaction between design, building, and testing than might initially be apparent. Even in relatively simple software, it's common to build a "quick and dirty" ***prototype*** that can be used to influence design decisions. As you see how that software works, you can modify it -- add features, change its behavior, make it more complex -- and "grow" the design. Eventually, you may learn enough about the software solution that you start from scratch and rebuild the system, but in building and testing you have learned what you needed about how to design this piece of software. As the software increases in complexity, these steps become increasingly intertwined, so that the actual implementation of the software development process may become more like an interactive community.

If the library system built for SmallTown succeeds, it's likely that the SmallTown library will want to extend its functionality. For example, the library might want to give its patrons access to the catalog over the web. This involves creating a web interface for a previously in-house custom piece of software. It might also be nice to allow a patron to reserve a book from home or even to ask the library to buy a copy of a particularly interesting book. These are wholly new functions not present in the previous system, but s that should be fully integrated with that system and take advantage of its existing data. These modifications and adaptations are an important part of the software life cycle.

Shifting requirements are a reality of software development. Sometimes, they are a result of an inadequately developed initial design. Often, they are driven by the changing world within which software is embedded. Consider the web browser. Originally, it was a relatively simple program for retrieving and displaying relatively simple (html) web pages. Over the first few years of web browsing, this requirement shifted slightly to encompass somewhat more sophisticated on-line material. Then, the web took off. Now, a web browser must support text and graphics, "plug-ins" (specialty programs that handle a wide variety of multimedia and other additional functionality), and even an interpreter for one or more programming languages (typically Java and JavaScript at this writing). The simple requirements of a mid-90s web browser have been transformed.

Spurred on by her amazing success with SmallTown's library system, SmallTown's software developer has been named Chief Technology Officer for newly formed Local Area Regional Library Consortium and spends most of her time meeting with government officials. She's recently gotten the Consortium members to create an aggressive five-year plan to create a joint regional library computer system based on the SmallTown software. A new software engineer--hired since the formation of LARLC--needs to understand how each the SmallTown software works; how it keeps track of the books in the collection and records patrons and checkouts; how it could be expanded to handle multiple collections at multiple (member library) sites; and whether any of the necessary adaptations for the Consortium will break the existing library system. Naturally, the CTO has no time to help him figure all of this out.

Although it is important to write code that works correctly, it can be even more important to write code that other people can understand. It turns out that far more time is spent revising existing code than writing new code. By writing your code well in the first place, you can make the job of maintaining and augmenting your code much easier.

Sometimes, you want to include information in your code that is intended only for people (e.g., readers of your code), not for the computer. You can do this by including whatever information you want in

comments. Comments are parts of your Java code that are not read by the computer, and the details of how to put something into a comment are included in the sidebar on Java Comments.

Documentation, or comments, are an important part of code-writing. Documentation is designed to help people read and understand your code. In spite of the running joke among overworked programmers, code is really not self-documenting. Learning to write good documentation may be even more important than learning to write good code. (This is particularly true since far more time is spent fixing, maintaining, and revising existing code than was spent in writing it in the first place.)

2.1.3 Software as a Process

The big question of this chapter is: How does software come to be? Earlier, we said that some people think of software development as a sequence of steps -- a recipe -- but that it is really much more like running a restaurant. To see this, let's think about the software development process as we just stepped through it and try describe the requirements of the software process. This will help us design a description of the job of a software engineer. We can use the idea of use cases from the previous chapter to help us think about this problem.

By far the most common use case for a software developer is a customer who has an existing piece of software and needs changes made to it, like the new hire who had to grow the SmallTown library system into a system for the library consortium. This often happens because the customer needs additional functionality from the software: to handle new kinds of information, produce new reports, work with another piece of software, run on a different computer platform, incorporate additional sites. Generally, the program was not originally developed by the person expected to modify it. Frequently, many people have worked on the software over time. Even understanding this kind of pre-existing ***legacy*** software can be a substantial task. (This task can be made easier if if proper care was taken by the original and subsequent developers to ***design for modifiability***, i.e., construct the program with future software developers and their potential tasks in mind.)

Other software development use cases are variants of these: fix (***debug***) existing software; build new software starting from two or more separate preexisting software components; understand a piece of software (e.g., to extract its principles for future use); test existing software in a new context; translate (or ***port***) software from one programming language to another or from one platform to another. Most of these use cases can be summed up in the notion of ***maintaining*** a piece of software -- keeping it functioning as ***bugs*** are discovered, hardware and software needs change, new functionality is needed, etc. Rather than an end result, there is an ongoing property to be maintained -- functioning software -- and there are a set of services available to modify, augment, and improve the software. By now, the job of a software engineer -- this collection of use cases -- should sound a lot like the restaurant model of processes. It is measured in terms of the ongoing adequacy of the software. The software itself must be cared for and developed much as a living, breathing thing. Thus, the name for this process *[@@@insert correct phases]* is ***software life cycle***. *[@@@stats]*

"Correct" software is a moving target. Requirements change. Software is never "done" for all time. Refinement is an ongoing process. Software should be designed, built, documented, and tested for ongoing improvement. Software built using the peanut butter and jelly notion of a correct answer -- code that is complete -- is applicable only when the produced code can safely be discarded after that use. (Even when you're certain this is the case, you'd be surprised how often it turns out that you need the software again.) Software that continues to be used will also continue to need maintenance and growth.

This repeated cycling through and between the various stages of specification (or design) development, implementation, and testing is a crucial skill for any good programmer. Classroom programs are too often written once and tested on obvious cases. Most of the time and money spent on real-world software is spent on revision and maintenance rather than on initial development. Acquainting yourself with this cycle -- and with writing clean, easy-to-read, reusable code -- may be the most important part of becoming a skilled programmer. These issues -- together with a tour through the development cycle -- are the main topic of this chapter.

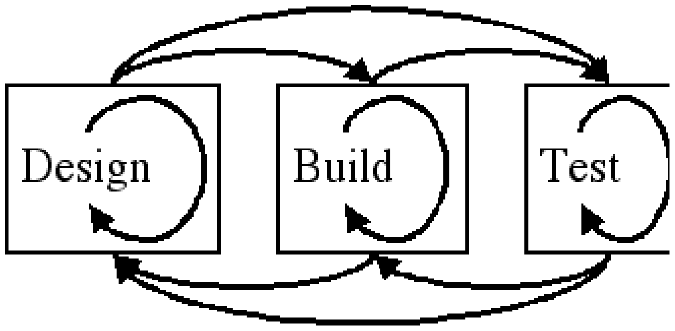

Figure 2.3. Design, build, test, always with feedback to earlier phases; each is an ongoing process.

2.2 Understanding the Problem

In the next few sections, we are going to step through the design of a library system like SmallTown's, including a computerized catalog and checkout system. Why a library system? First, libraries are probably things that you have experience with. The work of the library system we'll build is useful, but not too complicated. And this example illustrates many of the important stages and issues, so hopefully it will give you insights into what you should be doing and how.

We will use the questions of the previous chapter to flesh out the major portions of the design of this system. We will construct this system in English, not in Java, because this chapter does not presume that you know any Java yet. Since we don't have computers that run English, we can't actually execute the program that we build in this chapter. Also, there are aspects of the complete system that we will not get to in this chapter. Still, you should be able to understand how the program works by the time that this chapter is done.

We will also use the problem of designing the library system to explore the process of design and programming itself: understanding the problem, designing a solution, building the system, and testing its behavior. In the remainder of this book, we will explore the conceptual structures of which programs are built and their pragmatic implications. We will not spend much time, in the text outside of this chapter, looking at the larger process of developing software. There is, however, no way to learn to develop software without doing it. In the laboratories that accompany this book, you will have opportunities to build programs of your own. This chapter is intended to give you the context and background to apply what you learn in the remainder of this book to those laboratories and to software that you develop.

Later in this book, we will return to similar extended examples in segments that sit between chapters, called interludes. Each interlude focuses on a single extended example to illustrate the principles and practices described in previous chapters of the book and to ground them in a concrete example. In those interludes, but not in this chapter, actual working programs will be developed.

As you step through the various stages of program development -- understanding the problem, designing a solution, building the system, and testing its behavior -- you should keep track of your work, what you discover, and what you decide at each stage. You may want to do this in a physical notebook -- ideally a bound notebook, in which you do not remove pages -- or you may prefer to use a computer file. In either case, you should use this notebook to record things but not to erase or delete them. If you make a decision and later decide that it was wrong, it is important to preserve the original decision and the reasons behind it as well as the explanation of why you changed your mind. This is also a good place to record ideas you have about extensions or features you might add to the program or concerns you have about problems that might arise. You should date each entry. Whether it is a physical notebook or a set of computer files, we will refer to this as your ***engineer's notebook***.

2.2.1 What behavior do we expect?

Before we can build a program -- or even begin to design it -- we need to know what that program does. This is the "desired behavior of the program" question. What might we expect from a library's computer system? You should record your answer in your engineer's notebook

The primary users of the library system will be people like you and me who want to check out books. We will make requests of the library for books by specific authors, with specific titles, or on specific topics. We may know exactly what book we want, or we may need to find further information before we can select a book. Once we have identified the book that we want, we will need to check it out and later to return it. This library system has two main pieces: the catalog, which allows users to identify books, and the check-in/check-out system, which transfers responsibility for books between individual users and the library. The book itself is a physical object -- it will not be in our program, although some information representing it will be.

In more detail: If I want to get a book from this library, the first thing that I will probably do is to go to the catalog station and look up the book I want. This catalog should show me a screen that asks what I'm looking for. When I type in a ***query*** -- a request containing information about the book such as the title or author, for example -- the computer should display a list of books that match my request. I can select a particular book, and the catalog will tell me where in the library to find it (or whether it is currently checked out and so unavailable).

For example, I might ask for books about Alan Turing. The computer should produce a list of books that satisfy this property. If I decide that I'm particularly interested in *The Enigma* I can ask the computer for more information about it. The computer should indicate whether it is available to be checked out and where in the library I might find it.

Using the specific information provided by the catalog, I next need to retrieve the book I want. This part of the interaction involves physical space and objects, so it can't be handled by a typical computer; we'll see below several ways to solve this part of the problem.

Once I have the book in my possession, I need to give the library system my library card and the book. At this point, the library's computer needs to transfer responsibility for the book to my card. This includes updating its own records so that anyone looking for this book can discover that it is currently unavailable.

Other interactions with the library system involve returning a book -- checking it back in -- as well as variants on the above scenarios, such as identifying a book but then discovering that it's not available -- or extensions -- like adding the ability to reserve a book for future checkout. Another set of interactions -- on which we will not focus in this chapter -- involve library-maintenance functions, like determining which patrons have overdue books or adding new books to the system. In a real software engineering project, it is important to understand the scope of the project at the outset.

2.2.2 Use Cases

In this chapter, we will focus on the check-out and check-in interactions of a library patron. In order to understand better what they entail, we will flesh these out further as use cases -- particular patterns of interaction between a user and the desired system -- so that we can make them more precise. Each use case begins with an informal description of the interaction, which is used to clarify which interaction it is to both the user and designer. It also specifies the prerequisites of the use case -- what must be true in order for this use case to arise -- and its effects -- what changes occur as a result of the interaction -- as well as the sequence of actions and interactions that make up the activity of the use case. Of course, your use cases should find their way into your engineer's notebook.

1. **Title: Library-Lookup**
   I come to the library to look for book about jabberwocks. (My father told me to beware....)

   Prerequisites:
   I have (knowledge of) information/keywords that identify the book I'm looking for.
   Actions/interactions:
   I tell the computer catalog what I'm looking for.
   The catalog tells me which book I need.
   Effects:
   I know something I didn't know before; otherwise, this use case has no effects.

OK, so maybe that wasn't the most exciting use case. Let's do another one, this one including actually leaving with the book:

1. **Title: Library-Lookup-Checkout**
   I come to library to look for book about jabberwocks and check it out.

   Prerequisites:
      I have (knowledge of) information/keywords that identify the book I'm looking for.
      That book is available in the library.
      I have a library card.
   Actions/interactions:
      I tell the computer catalog what I'm looking for.
      The catalog tells me which book I need.
      I get the book and check it out.
   Effects:
      The book is transferred from library's possession to mine. (Better record this fact somewhere!)

Note that this use case is really the same as the previous use case with some extra actions/effects added onto the end. In fact, the second half of the lookup-checkout use case could be its own use case, if I came to the library already knowing precisely what book I was looking for.

@@ *see exercise* # @@

Of course, the **Library-Lookup-Checkout** use case presumes that the book is available. If it is not, a slightly different use case results:

1. **Title: Library-Lookup-Can't-Checkout**
   I come to library to look for book about jabberwocks and check it out.

   Prerequisites:
      I have (knowledge of) information/keywords that identify the book I'm looking for.
      That book is not available in the library.
      I have a library card.
   Actions/interactions:
      I tell the computer catalog what I'm looking for.
      The catalog tells me which book I need.
      I try and fail to get the book. *(OR the catalog tells me it is not available.)*
   Effects:
      I know something I didn't know before; otherwise, this use case has no effects.

With all of these books being checked out, we should also include a use case to return a book to the library:

1. **Title: Library-Checkin**
   I return a book to the library.

   Prerequisites:
   I have a particular book in my possession. (The library has a record that this book is in my possession.)
   Actions/interactions:
   I give the book to the library.
   (The library records this fact.)
   Effects:
   The book is transferred from my possession to the library's, presumably updating the library's record system.

There are, of course, other possible use cases for a library, even one as simple as this. By now, though, the basic ideas should be clear. Each use case lists its prerequisite conditions, its actions or interactions, and the effects it has. For each use case, you can design a test of the system you eventually build; that test will verify that the system supports the desired behavior. For example, Library-Lookup-Can't-Checkout could be tested by ensuring that *Alice in Wonderland* is not available and then asking the system to check out all books written by Lewis Carroll to a particular patron.

These tests form the beginning of a ***test suite***, the set of tests that you develop along with your program and that you will use to ensure that your program behaves as it should. As you develop them, these tests should go in your engineer's notebook along with the use cases. For each test, be sure to record its inputs, the timing of those inputs, and the behavior that you expect to see. You can also record any signs that you ought to see -- part way through -- that things are going right as well as signs that indicate a problem. Developing tests along with -- or even before -- your design is an indication that you understand the integrated nature of the software life cycle.

2.2.3 Assumptions

Use cases document the way that a system is intended to be used. Behind these use cases are a series of assumptions. It is always important to make these assumptions explicit and to record them. Many decisions that you as a software developer will make are based on your assumptions. It is essential that you understand the assumptions that you are making and that you check to see that these assumptions are really valid. What assumptions do we make about our library?

Some assumptions reflect operating conditions, i.e., when the program can reasonably be expected to behave properly. For example, we are going to make some strong assumptions about correspondence between what is in the on-line world of our program and what is going on in the real world around it. If (the electronic information corresponding to) a book is checked out to (the electronic information corresponding to the library card of) a patron, we assume that the physical book is in the possession of the appropriate patron. When the book is listed as in the library, we assume that it is in fact there and appropriately shelved. All of these assumptions are likely to be violated by a real library -- in which books are sometimes stolen or mis-shelved -- but we assume that violations are addressed outside of the scope of the program we're designing. (We might want to think about how someone could manually override parts of the program to correct these issues should they arise, though.)

In almost every program, there are things that are outside the scope of that program. Being explicit about the operating assumptions of the program helps its ultimate users understand what the program can and can't be counted upon to do. For example, in this case a librarian might periodically take inventory of what is actually on the bookshelves. This kind of human check on information collected by a program is especially helpful to maintain a correspondence between the online and real world. This assumption, together with, e.g., an assumption that cards won't be forged, are really assumptions about the way that this system is embedded in a larger society and about social practice within that society.

To make our implementations easier, we will assume that each book is equipped with a unique bar code and that the checkout and checkin is performed using a bar code reader. A bar code reader is a piece of computer equipment that can read the bar code on an object and produce a number corresponding to that code inside the computer.

We will also assume that every library patron has a unique library card with its own unique bar code, and that library card is a good stand-in for the patron. We will not, in the system that we are designing here, explore how people are given library cards; this is a separate system that we could build. We will assume that we don't have to worry about forged library cards.

This assumption raises questions about where unique identifiers such as the bar code come from. In fact, there is an additional assumption hidden here, that every library card and every book has a unique bar code. This assumption has to be enforced by a human being (or another computer program) whose responsibility it is to distribute bar codes. If this assumption is violated, the program will not be able to tell which book is checked out to whom. Enforcement of unique identifiers isn't too hard if there is only one centralized place where they're given out, but what if each branch of the library is allowed to assign bar codes to library patrons? How do they make sure not to assign the same number in two places? There are some straightforward ways to deal with this -- assign each branch its own initial sequence that's part of every bar code it issues -- but in general the question of assigning unique identifiers is a complicated one in a distributed system.

Other assumptions may help in simplifying program development, but might eventually be relaxed. Initially, at least, we are going to assume that the library has only a single copy of each book. This is a potentially dangerous assumption, as it lets us treat the identifying information of a book (author, title, etc.) as interchangeable with the book itself. We will give each individual book its own bar code, but we will also assume that there is only one bar code corresponding to a particular title/author combination. That way, we won't have to worry about two books with different bar codes that are otherwise identical. By making ourselves aware of this assumption explicitly, we can plan for a future version of the system in which there might be additional information determining which of several interchangeable copies of the book we have.

If we were to relax this assumption, we would need to add a new component to the system to manage the multiple interchangeable copies of a particular book. If we design the system carefully, we can later plug such a component in without disrupting the whole system. It's important to know where this assumption matters -- in the catalog -- and where it doesn't -- in the checkout system, where the unique bar code of the book is all that matters.

Another simplifying assumption involves the role of time. We are going to assume that actual time doesn't matter in our initial prototype. We might choose to time-stamp check-out transactions, but we won't worry about how much time has passed, whether books are overdue, or other such issues. In other words, we will not worry about the accuracy of any clock in the system. Even if we eventually decide to add information about due dates, fines, etc., we can do so without having to worry about actual times; we'd only need to keep track of dates.

Some systems, like the library system described here, can have very relaxed notions of time. Other systems, such as a robot controller or an automobile's cruise control system or a microwave oven, need to function in real time: time matters to every aspect of their operation, and time inside the system must be locked to time outside the system. Most systems fall somewhere in between, needing to keep track of time to some extent, but not to be exactly in lock-step with the rest of the world. For example, a hospital's pharmaceutical inventory control system needs to keep track of who got what medicine when, but it is probably not important for inventory control to be accurate to within ten minutes, and it certainly doesn't have to be accurate to within seconds or milliseconds.

We will assume that the programs running this system are robust and that the computers on which they run do not crash in the middle of things. This is definitely a bad assumption -- computers do crash, and a program such as this really does need to be secure even if the computer system crashes in the middle of a check-out. A real-world program of this sort would need to contain extra machinery to deal specifically with this problem. Our version here will not address these issues.

Finally, there are assumptions that we do not make. These non-assumptions should be recorded in your engineer's notebook as well. We will not assume that there is a single check-out point. Instead, we assume that two different people could check two books out simultaneously, and that our system has to be able to handle this. As with unique identifier distribution, this imposes additional complexity on our system. It means that whatever structure keeps track of who has which book will need to be careful not to let the same book go to more than one person, or be simultaneously checked out to a patron and in the library. The issues raised by this assumption are addressed in some depth in chapter [Concurrency].

2.2.4 Promises/Guarantees

In addition to understanding the assumptions our programs make, we need to spell out the promises or guarantees that are a part of the behavior they will provide. These form a basis for the contract our system will make: what you need to know to interact with the system. We will see later that individual component elements of the system may also have their own promises and guarantees -- their own contracts -- that they make to one another. In all cases, promises and guarantees are important to record in your engineer's notebook.

When a user asks the catalog about a title, author, or other characterizing information, the catalog promises to include all matches in the information it supplies the user. Said another way, if a book matches a user's request, that book will be included in the answer provided. We should also include the opposite promise: Only those books that match will be included in the answer. Otherwise, the catalog could simply list all of the books in answer to every question; this would meet the first of these promises (if a book matches, it will be listed), but wouldn't be particularly useful.

The first of these promises -- if a book matches, it will be listed -- is called ***completeness***. It means that the system contains (or supplies) all of the (true) information. The easiest way to guarantee completeness is to have the system supply all information, true and untrue, relevant and irrelevant. The second promise -- only matching books will be listed -- is called ***soundness***. It means that the information in the system is correct. The easiest way to guarantee soundness is to have no information in your system. An ideal query system -- one that lists exactly those books that match -- is both sound and complete. [[Footnote: Technically, a system is either sound or unsound, either complete or incomplete. It is often useful to talk about how well a system matches these criteria, though, and there are different terms used to describe these properties. ***Recall*** is the term for how close a system comes to being complete. The hypothetical "return all books" version has perfect recall. The term for how accurately the returned suggestions match -- how closely the system approximates soundness -- is ***precision***. The return-everything version has very poor precision. A return-nothing version has perfect precision -- all of its nonexistent suggestions are matches -- but lousy recall. The problem of optimizing precision and recall simultaneously is the subject of the field of ***information retrieval***.]]

Why would we want to relax either of these promises? Perhaps the user will issue a query that matches 100 books, or 1000. Do we want to display all of these? Is it OK to display only a set, or to tell the user that the query has too many matches? Perhaps. We won't implement these features in this chapter, but they are extensions you could imagine adding to our system.

Alternately, maybe we want to include some "near misses", books that we think the user might have been asking about even though they don't strictly match. For example, if the user asks for books about "Harry Porter", we might want to suggest J.K.Rowling's Harry Potter series or even the business wisdom of Michael J. Porter.

Again, we will not explore these extensions in this chapter, but they are certainly reasonable add-ons one might pursue. Both involve additional sensitivity to the needs of the human user. We will begin to touch on the issues of user interface design in this book, but the field is one that you will want to learn more about as you develop your software engineering skills.

Another set of guarantees involves the checkout system. We need to ensure that every book is in the possession of exactly one patron, or of the library, at any given time. A book cannot simultaneously be in the possession of more than one of these parties. It wouldn't do to have the system check out a single book to two patrons, or simultaneously list the book as checked out and on the shelf!

This means that when a user checks out a book, that book will be associated with that user's library card and NOT with the library itself. When the book is checked back in, it is cleared from the user's library card and associated with the library again. Further, a book must be in the possession of the library before it can be checked out. So if a book is checked out to a patron, it cannot be checked out to another patron until it has first been returned to the library. Guarantees such as these suggest additional tests that you will want to add to the test suite emerging in your engineer's notebook.

Remember, this is a guarantee about the record-keeping inside the library system, not about the physical book. We assumed above that the physical book would be in the right place, but the computer can't guarantee that itself.

Finally, we want a guarantee that whenever the book is in possession of the library (according to the checkout/checkin system), the catalog lists it as available. Again, this is a two-way condition: the book should be listed as available whenever it is in the library's possession and *only* when it is in the library's possession. Otherwise, we could always list it as available to satisfy the first promise. We will actually be willing to allow a gap of as much as a few seconds between when the book is checked in and when the catalog lists it as available, but it would not be OK if this gap became hours, or days. This is a constraint on how quickly information has to get from one part of our system to others.

2.2.5 The Community Around You

No implementation happens in a vacuum. The assumptions and guarantees of the previous section are to be met by a system that you will build out of existing parts and to interact with an existing environment. Assumptions document constraints that your implementation will impose on the environment. Guarantees are constraints that you've agreed to let the environment make on you. These two sets of requirements form a part of the specification of the interface between your system and its environment. If written well, they may be all you need to know about the world in which your system will be embedded.

But there is a second way in which your system is affected by things around it. Invariably, you will use tools that you didn't build to construct your system. Some of these -- the Java language, for example -- are very general purpose and you will use these over and over. Others are more specialized and you will only use them on a particular occasion. For example, if your job is to upgrade an existing system to work in a new context, that old system may form a piece of your community. You will also likely find that many specific problems that you encounter have ready-made solutions that someone else has built. Being able to find, understand, and incorporate other people's tools into your systems is an extremely important skill.

In this section, we review some of the major elements of the communities in which your system is embedded.

2.2.5.1 Physical Environment

We have already seen how the assumptions and guarantees of the library system constrain the kinds of environments in which the system would work. Some of these constraints may have come from our desire to simplify the implementation: We are not (yet) building a system that provides internet access to the library, for example. Other constraints may come from the real world requirements of the customer for whom we are building the system: Multiple librarians need to be able to use the system simultaneously.

When you are building a piece of software, you need to understand the requirements of the customer (or the problem definer) before you begin. This can take some back-and-forth as you propose solutions, the customer decides that you've misunderstood or realizes that s/he has an additional need your system won't meet, and you revise your proposals in response. There are also real constraints--the types of computers available, the specifications of components you need to integrate--that are non-negotiable. For example, the library may already have purchased bar code readers; you will have to work with their actual interface.

2.2.5.2 Program Libraries

It is extremely rare to build an entire system from scratch. [[Footnote: In fact, even "from scratch" usually relies on program-building tools that already exist, but in this case we're talking about using pieces of programs that others (or you, previously) have built.]] Usually, your problem decomposition will eventually turn up the need for some components that already exist.

For example, our library checkout system will make use of some pre-existing components. Of course, we'll assume that the bar code scanner reads a bar code from a book or from a library card and produces a number. We'll want to be able to use that number to identify the computer's ***record*** for a particular card or book. [A record is just the computer's representation of information about that real-world object.]

To accomplish this, we will assume that we have a pre-existing piece of software that can associate a ***key*** -- like the bar code number -- with a ***value*** -- like the computer's record of a particular book. In fact, this software component should be able to store a large number of keys and their associated values and respond to any key by supplying the associated value. This particular kind of structure is called a ***lookup table***. It is a very common kind of software component, and Java provides several different kinds of lookup tables, as we shall see in later chapters.

Pre-existing software, such as the lookup tables provided by Java, is often collected into groups of inter-related components. These components aren't complete programs by themselves , but they are frequently useful in building other systems. (A lookup table is a good example; it's not usually of much use until it's incorporated into a larger system.) Such a set of program components that is sitting around, already written and waiting to be used, is called a ***program library*** (or, when we're not also talking about the book kind, sometimes just a library). Java has a number of very useful libraries that are part of its standard distribution, but programmers all around the world produce a much wider variety of libraries. Many of these are available on the web. Many are freely available; others require the purchase of a license. In this book, we make use of a set of libraries (the "cs101" libraries) that are freely available and were designed specifically for this curriculum.

Figure 2.4. A lookup table remembers associations between key and values. It supports two actions: put (a key with a value) an get (the key associated with a value).

In building our checkout system, we are going to assume that we have several already-filled-in lookup tables. For example, we'll assume that we have a lookup table that associates individual book bar codes with generic book descriptions, such as authors or titles. If we were really building this system, we'd need to supply a piece of software that allows a person to enter information about new books. This way, the lookup table can be created or extended as the library grows. A similar lookup table relates library card bar codes with information about library patrons (like the address to send their overdue notices!); a complete system would also allow a way to add a new library card and the patron's information.

We will also need a piece of software that can identify any records that share a particular field. This will be used to identify, for example, all book records that share a particular author. This can be accomplished with a set of key-value lookup tables, but there are also other ways to build such a system. We won't worry about how that component works; we'll just assume that we have one. This lets us get from a query about books written by Lewis Carroll to *Alice in Wonderland* and *Through the Looking Glass*.

Finally, we will assume that we have a number of components that present information to the user nicely and elegantly. These components may use windows, icons, menus, etc., to facilitate the user's interaction with various computer screens. We will begin to explore how such things might be implemented in part 4 of this book, but for now we will simply describe what information needs to be presented to the user or obtained from her, without specifying exactly how that information should appear on the screen.

2.2.5.3 Users

In building our system, it is important to remember that not all of the members of the community are program libraries and physical devices (such as bar code scanners) and other pieces of software or hardware. Our systems frequently involve interaction with human beings. People have a set of requirements that differ from hardware and software. People are much more adaptable to your system; their requirements are often more flexible. But good computer systems also make things easier for human users. If a person sits down to use your system, s/he should not have to read a thick manual before getting started. A good system design will incorporate the natural abilities of a human user so that the system is intuitive to use.

The part of your system that interacts with humans is called the ***user interface***. Human community members, like hardware or software community members, come with their own sets of assumptions and guarantees and you will need to design an interface that works for human community members just as you would for other members of the community or environment around your system. Human beings typically appreciate visually presented information (though sometimes it's important to use other modalities, such as sound). Humans benefit from clear labeling and instructions that would be superfluous for a machine. Human time scales are typically slower than machines -- responses are measured in *hundreds* of milliseconds -- but people are much less patient than machines when delays become long.

A good understanding of how people work is the goal of the field of ***human factors*** analysis. Because interaction with human beings is an important part of many computer programs, every computer programmer should learn how to design a good user interface. Many of the properties of good interface design are obvious. A user interface should be simple, clear, intuitive. It should make it easy to do what you want to do and harder to make mistakes. For example, when you insert a new software CD in your computer, you may have to spend five minutes locating the install file. Alternately, inserting the CD might immediately open a window that says, "Do you want to install this software now? If so, click here to begin; if not, click here to close this window." The second is a much more intuitive interface for such a CD. But many aspects of user interface design are more subtle or not given sufficient attention by system designers. Otherwise, how can you explain the difficulties that so many people have in programming a VCR, using a new computer program, setting up a printer. Too many programmers have focused on designing for the physical, hardware, and software components of their environment and paid little attention to the human beings who are their system's most important community members.

Sometimes, a system will have more than one user interface depending on who might be using it. A simple version of this is a web site that offers a frames-and-images version or a very simple text only version of the same information. In a system like our library checkout system, we might provide one clean, simple, intuitive interface for the librarian and/or library patron and another -- more obscure, more complicated, less intuitive -- interface for the system programmer who is responsible for maintaining the checkout system.

2.2.5.4 Understand their interfaces (and assumptions)

When you commit to using or interacting with something from the community around your program, you need to understand its behavior, assumptions, and guarantees. These amount to its contract -- the behavior it promises to you and the circumstances under which it makes these promises -- as well as its peculiar properties.

For example, people have very flexible behavior and somewhat negotiable interfaces. You can train a human user to interact with your system in a particular way. (Just think of all of the crazy things people do to get their computers -- or other machines -- to cooperate. [[Footnote: Donald Norman has written an excellent book on this subject, called *The Psychology of Everyday Objects*.]]) In fact, it is generally much easier to change the behavior of a human being than that of a computer program. However, people have some particular expectations that are not really negotiable. A computer may be willing to wait minutes at a time for an answer; in many cultures, a person is rarely willing to endure delays that are measured in seconds.

Physical environments tend to be much more rigid than people are. But a program can be artificially constrained to work only in particular physical environments: most robots only work indoors, not outdoors, and most wheelchairs cannot go up or down steps. A physical environment can also sometimes be modified, tailoring it to your program: curbs can be cut to make ramps, or books can be outfitted with bar codes that uniquely identify each one. Sometimes, you can even find regularities in a physical environment that you can exploit to make your program work better, like the fact that the title of a book is usually the set of words printed in the largest type on its title page.

Code libraries, when you use them, should generally have good documentation that explains what that set of code does and under what circumstances. In this book, we will explore some existing code libraries in Java and you will learn how to understand what they may be able to do for you. New tools are constantly being created, though, and you will need to build the skill of understanding a new library. Software engineers rarely build from scratch. You should always be on the lookout for good tools that can help make your job easier. As you encounter them, record them in your notebook along with the problems that you think they might someday help you solve.

2.2.6 Requirements are a moving target

There are many additional features that one can imagine adding to the system as described here. For example, it would be nice to have another part of the program that could look over all of the books checked out of the library and determine which of these were overdue; overdue notices to the corresponding patrons could then be generated. We are not going to design this feature in now, but thinking about it reminds us that the check-out transaction will need to be date-stamped; that is, we'll need to know when it happened (or at least when the book is due to be returned). This kind of anticipating possible future augmentation often turns up modifications to the basic system. Not all of these should necessarily be accommodated -- simplicity is an important principle -- but thinking about them can often help you create a more robust base system.

In this book we will rarely talk about *the* solution to a problem, as though there were only one. Instead, we will explore many ways that particular problems can be solved, and we will compare and contrast these different approaches. Although there is rarely only one right way to solve a problem, there are invariably wrong ways, as well as less desirable ways, to do it. In this book, you will not only learn about useful techniques. You will also develop some of the judgement that a skilled programmer needs about which approach to use when. Of course, this judgement is something that you will continue to develop through your experience writing, understanding, testing, and modifying programs.

The moral of this story is this: A good specification makes program development easier. However, it is a rare specification that is definitive. Instead, most program specifications are representations of the designer's understanding of the problem to be solved at a particular point in time; every program should be built with the understanding that it is likely to grow and change in often-unanticipated ways. Part of good program design is building something that works and meets the specifications set out for you. Part of good program design is understanding and developing those specifications, including directions in which they might actually change in the future. And part of good program design is developing programs that are easy to understand and modify, documenting not just what your program does but also why and how you made the design decisions that you did, so that it will be easier to modify your program in the future.

Above all, the overriding principle of design is not to unnecessarily complicate systems, especially in the early stages of design. The more streamlined and simple the core of your system, the more likely it is to be able to accommodate unanticipated changes because it will be easier to understand and work with.

2.3 Designing the System

In the previous section, we asked what behavior our program should have. In this section, we will begin to decompose that behavior into the pieces -- the community members -- whose combined efforts will create that behavior. The questions that we will look at in this section are:

- Who are these members of the community? What entities combine to create that desired behavior? This includes an understanding of the desired behavior of each of these entities in turn.
- How do these community members interact? What contracts -- what behavior, assumptions, and guarantees -- do they make with one another?
- What goes inside each one? That is, what is each of these entities made of? Is it, itself, a community? If so, we will need to ask the questions of this section about it, as well. Or is it a simple instruction follower? In this case, we will need to write its recipe. Or is it something outside the system that we are constructing -- a program library, a human being, a physical system like the bar code scanner -- whose properties we need to understand but whose implementation we can simply adopt?

In asking and answering these questions, we will develop the implementation -- the solution -- to the problem requirements we described above. Like the requirements exploration, the solution design is an important piece to record in your engineering notebook. Begin with questions and proceed, step by step, to answer them.

2.3.1 Who are the members of the community?

One way to figure out what things your program needs is to look for nouns and verbs. That is, in your description of the system, you will talk about the things that are a part of your program and the actions that they perform or are performed on them. The things -- the nouns -- are objects or entities that you will likely need to create. The actions are recipes that these things will follow. In a library, typical nouns include book, library card, and catalog; typical verbs include check in, look up, etc. We'll begin with the nouns.

Consider, for example, book. There is a physical thing -- a book -- in the world, but inside the computer we're going to need some other thing to represent the book. Java, the language we'll ultimately be using to build our programs, is a kind of language called an ***object oriented language***. This means that most of the things in a Java program -- the nouns, the stuff, the bits and pieces that are manipulated -- are objects. (An object is a particular kind of computer structure, about which you will learn more in part 2 of this book. For now, *object* is how you say *thing* in Java.) So, in Java, we will create a particular kind of object to contain all of the information about a physical book that we want to represent in the computer. We'll call that object a BookID. (There's nothing magic about this name. You could call it a Fred or a Football, but your program might be harder to understand in that case.) A BookID might contain the title and author of the book; it will certainly need to include its bar code.

Another noun for which we'll need a kind of object is the library card. We'll call the electronic object that contains all of the information about the patron whose card it is a PatronID. When a book is checked out to a patron, we will record this connection between the book's BookID and the patron's PatronID. We will even create one special PatronID for the library itself, because it will be convenient to be able to treat books that are not checked out as being associated with the library's PatronID. We will give the library's PatronID a special name: LIBRARY_ID. [[Footnote: Again, there's nothing particular about this choice of names except that we think it will help us remember. If you'd rather, you can call the library's PatronID Rumpelstiltskin.]]

The card catalog is the old-fashioned place that you would go to look up a book. Now, it's more common to use a computerized version that operates more like a web search engine. For this quaint historical reason, we'll use the name CardCatalog for the kind of object that keeps track of what books which author wrote. Of course, our CardCatalog isn't likely to contain any of those lovely old cards, but it will hopefully be faster to search.

A CardCatalog is the kind of object that you'd like to be able to ask to do something for you. In particular, you'd probably like to be able to make requests like "look up Shel Silverstein" of your CardCatalog. In an ideal world, you'd ask the CardCatalog to look up something, and it would hand you back a piece of information uniquely identifying a particular book -- a BookID. So we'll assume that the CardCatalog has a thing that it knows how to do, called lookup. Note that this is a funny kind of assumption, because we will have to *create* the recipe for doing lookup later. For now, though, we're saying that CardCatalog will have such a recipe, and not yet worrying about how to implement it.

This is a completely typical way to do design. We are asking, "who are the members of the community?" We are presuming a (partial) answer to the question, "how do they interact?" -- CardCatalog will provide a lookup service -- but not yet worrying about "what goes inside?" We are not done until all questions are answered, but we don't have to answer all questions at the same time.

What other nouns does our system need? Well, somewhere there is a record of who has checked which books out. For similarly quaint historical reasons, let's call the computer representation of this record a CirculationDesk. There are three verbs associated with the CirculationDesk: check out, check in, and verify availability.

In a real library, there is another thing that comes between the card catalog -- the place where you look up the particular book you want -- and the circulation desk -- where you transfer responsibility for the book from the library to yourself. This is the bookshelf, where you go from the description of information that the card catalog gives you to the actual, physical book. Because the actual physical book isn't inside the computer, our system may not need to include a component to deal with it. But if there were something that the computer needed to do with the physical book, Bookshelf might be a kind of object in our system and its associated verb would be fetch the book.

Note that BookID and PatronID don't have associated verbs in the above description. So far, we have no need for either of these kinds of objects to do anything in particular. But if we were going to send overdue notices out, we might, for example, want PatronID to have an associated print mailing address verb. These kinds of things can be added to the system now -- at design time -- or later, as the system continues to grow and improve.

2.3.2 How Do They Interact?

Once we know what kinds of things exist in our system, we can write down how those things work together to create the overall behavior of that system. If the "Who are the members?" question is really about nouns, "How do they interact?" involves looking closely at verbs. For each, we specify what inputs it needs, what outputs it provides, and under what assumptions it operates; in short, what contracts it makes. Putting these pieces together should yield your use cases. As always, your notes from this step are fodder for your engineer's notebook.

When I ask the CirculationDesk to check out a book, I need to supply it with two items: the BookID of the book I want checked out and my PatronID. So a CirculationDesk's checkOut recipe needs to be supplied with a BookID and a PatronID. I might, for example, say:

circulationDesk, please checkOut this *bookID* to *patronID*

where I'd need to specify a particular *bookID* corresponding to the book I wanted to check out and the *patronID* of the person to whom the book should be checked out. (I could, for example, get those two pieces of information from the bar code reader scanning the book and the library card in question.) It might also be a good idea for the CirculationDesk to let me know whether this checkOut succeeded. So the contract for a CirculationDesk's checkOut action is: needs a BookID and a PatronID, provides a signal of success or failure.

The formal part of a contract says who is offering this behavior (or at least what types of "who"s), what that entity needs to be given, and what it provides in return. The informal part of a contract -- often included in accompanying documentation -- specifies the relationship between what is given to this entity and what it returns, what else changes while the contractual behavior is happening, and when the contract can or should be used. Although a legal contract is generally made between two parties, a software contract is really offered by one (kind of) entity and can be used by anyone willing to agree to its terms.

The checkIn contract is not quite the same as checkOut. After all, when I'm returning a book, I don't need to specify a PatronID. So checkIn requires a BookID and provides a signal of success or failure:

circulationDesk, please checkIn this *bookID*

The CirculationDesk's verifyAvailability action needs a BookID and provides a yes/no answer.

The CardCatalog's lookup action is more complicated. I would like to be able to give it an author or a title or a keyword or several of these things at once. I might want to distinguish these as CardCatalog lookupAuthor or lookupTitle actions, or I might want to hide all of that machinery inside the CardCatalog object and just have one lookup action. There is not a right answer to this question; there are advantages and disadvantages to designing this object in either way. One important point, though, is that a user who is expecting the CardCatalog to provide a lookupAuthor action is going to be very surprised if the CardCatalog only has a lookup action (or vice versa). So even when the decision may seem arbitrary, it is important to make the decision and to document the decision so that all of the pieces of your program can work together. For the purposes of this chapter, we will imagine that CardCatalog simply has one action, lookup, and any magic about titles or authors is handled by the CardCatalog out of our sight. (Of course, this makes the job of the designer of the CardCatalog harder.)

At this point, we can begin to piece together scripts corresponding to each of our use cases. For example, a script for the Library-Lookup-Checkout use case might look something like this:

1. Patron enters library, approaches CardCatalog.
2. Patron asks CardCatalog, "please lookup books about jabberwocks"
3. (CardCatalog does its thing. Fleshing out this step requires looking inside the CardCatalog, which we'll do below.)
4. CardCatalog tells Patron the BookID associated with *Alice in Wonderland*
5. Patron [[Footnote: or, in another possible implementation, CardCatalog!]] provides this bookID to CirculationDesk along with PatronID, asking, "please checkOut this bookID to this patronID".
6. CirculationDesk carries out its action (again, using a script to be written below) and reports success.
7. Patron departs, happy.

In this case, the script is just an embellished list of the actions involved in the use case along with a detailing of the information provided by each community member to another, i.e., how one entity uses the contract of another. If you were to act this out with multiple people, this script would be sufficient to describe all of the actions of the library patron. Eventually, both CardCatalog and CirculationDesk will need to have scripts -- recipes or playscripts -- of their own so that they can carry out their parts of this drama. We will turn to these questions next.

2.3.3 What is each one made of?

So far, we have decomposed the library into interacting community members like CirculationDesk and CardCatalog. Now, we turn to each element and examine what goes inside: what is it made of? When we described each community member, we asked what its behavior was and what contracts it made with other community members. Now, we ask how we can accomplish this. Is the community member itself a community, or is it a simple instruction follower? If it is a community member, we must ask the community questions -- who are *its* members, how do *they* interact, and what is each one made of -- all over again. If the community member is simply an instruction follower, then we must write its recipe.

Let's now look at some of the individual components of our library. We'll start with the CirculationDesk.

2.3.3.1 Some Decompositions are Communities

A CirculationDesk has to provide three actions:

- checkOut (a BookID to a PatronID)
- checkIn (a BookID), and
- verifyAvailability (of a BookID).

If these actions are to be requested by other pieces of the system, they will need to be provided as pieces of program code that can be run, or ***called***, by other pieces of code. (This is like when one recipe refers to another recipe in the same book.) This collection of callable services is called an ***interface***, and you will learn much more about interfaces in chapter 4.

The CirculationDesk actions might also (or instead) be requested directly by a human being. In this case, there ought to be some machinery that makes it easy for the human being to make this request: a user interface. For example, part of the CirculationDesk might include a bar code scanner with a "check out" button attached; pressing "check out", then scanning a library card followed by one or more books would be regarded as a request to check out those books to that library card. The whole CirculationDesk would consist of two sub-entities: the software that operates the bar code scanner and makes requests, and the piece of software that provides the checkOut/checkIn/verifyAvailability interface described above.

Or the CirculationDesk might put up its own web page, allowing a person to type in the bar code number of the book s/he wanted to check out as well as a valid library card bar code number. The web-page-providing piece of the CirculationDesk would be a different kind of user interface. The web page controller would know how to make things show up nicely in a variety of different browsers. In addition, like the bar code scanner-and-checkout system, the web page controller would know how to make requests of the same old piece of the CirculationDesk that provides the checkOut/checkIn/verifyAvailability interface.

Once again, there is not a right way to make these design choices. However, since all three designs involve the same checkOut/checkIn/verifyAvailability interface, it makes sense to focus early efforts on that piece of the CirculationDesk and only later to add one or both of the other functions. (See Keep It Simple and Keep It Working, below.) In this case, the promise to the other parts of the system you are building is the same, so the availability of the different user interfaces should not affect the working of the software other than the CirculationDesk.

Figure 2.5. @@pics of different CirculationDesk decomposition co/ci/va function alone (providing interface to rest of code); ba code scanner ui and co/ci/va function; web page and co/ci/va function; all three coexisting..

2.3.3.2 Other Decompositions are Recipes

And what goes inside the checkOut/checkIn/verifyAvailability part of the CirculationDesk? We have said above that we are assuming the presence of a lookup table, a piece of pre-existing software that will associate a key -- like a BookID -- with a value -- like a PatronID. Java provides several possible structures suitable for this purpose, as do most modern programming languages. Let's call the particular lookup table inside our CirculationDesk the masterList. The masterList, like any lookupTable, has two main actions:

the masterList can put a *bookID*, *patronID* pair into its records

the masterList can get a *bookID*'s record, which provides the associated PatronID

Once we have the masterList to work with, it's not that hard to figure out how to write the recipes for the CirculationDesk functions. For example, CirculationDesk checkOut could be implemented as:

to checkOut a *bookID* to a *patronID*:

1. masterList, please put the pair *bookID, patronID* into the record
2. report success

The first step of this recipe is simply an invocation of masterList's own put recipe; the second step completes the checkOut contract.

Similarly, we can build recipes for CirculationDesk's other actions in terms of masterList's actions:

to checkIn a *bookID*:

1. masterList, please put the pair *bookID,* LIBRARY_ID into the record
2. report success

recalling that LIBRARY_ID is the special PatronID assigned to the library.

to verifyAvailability of a *bookID*:

1. masterList, please get *bookID*'s record; call the associated PatronID whoHasIt
2. report availability exactly if whoHasIt is LIBRARY_ID

In other words, look to see who is recorded as having the book, and say it's available if this answer is "the library."

The decomposition of the CardCatalog is very similar in principle to CirculationDesk, relying on a structure that keeps track of keyword associations to BookIDs. However, since the lookup needs to be able to happen in several different ways -- by author, by title, by keyword, etc., and by various combinations of these things -- the underlying record-keeping is likely to use a more complicated structure than a simple lookup table like masterList and the associated recipes are likely to be somewhat more involved. Additionally, the CardCatalog may have a very sophisticated user interface, allowing easy presentation of large amounts of information, or provide support for incremental refinement of search criteria as the user tries to narrow down what s/he's looking for.

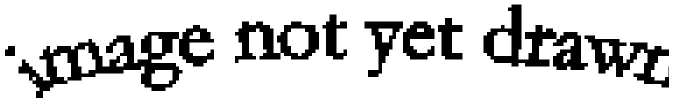

Figure 2.6. @@ Draw pics of these fns and their interactions: master.get(bookID) -> patronID; master.put!(bookID, patronID); circulation.available?(bookID)->T/F; circulation.checkOut!(bookID, patronID); circulation.checkIn!(bookID) @@.

2.3.3.3 Community Members Come in All Shapes and Sizes

You may recall that there was a third member of the library community, invisible in our implementation but potentially important in other versions. This is the Bookshelf, which would fetch the book.

In our implementation, we presume that the human being in the library is responsible for bringing the BookID supplied by the CardCatalog's lookup to the CirculationDesk for checkOut, presumably stopping by a physical shelf to pick up a physical book along the way. So Bookshelf's fetchTheBook action is actually performed by a human being -- the library patron -- in this case.

The informal system just described actually corresponds to a formal system in place in certain ***closed stack*** libraries, such as the United States Library of Congress or the New York Public Library's main branch. In a closed stack library, individual patrons are not allowed to wander into the stacks of bookshelves and select books for themselves. Instead, a request for a precise book -- essentially, a BookID -- is given to a staff member of the library, whose job it is to go and fetchTheBook from the closed shelves and provide it to the patron. Our library system would work just as well in this kind of a setup, provided that the BookID produced by the CardCatalog's lookup action was given to the library staff. Again, though, the work of this recipe is being done by a human being; in this case, a library staff member.

A mail order library or circulating collection, such as Books On Tape (TM), works similarly. You request a particular item from the library, and fetchTheBook is implemented by the employees of the mail order company, often in conjunction with the post office or parcel service. In this case, the behavior of this system component may be provided by a literal community -- the various order-takers and shelf-pullers of the Books On Tape corporation plus a fleet of trucks or planes and associated personnel at the delivery service -- in order to fulfill the same behavioral interface.

A fourth alternative implementation involves an eBook, in which there may be no physical object to be transmitted. In this case, the Bookshelf's fetchTheBook action might involve a lookup table like the one used by the CirculationDesk. This lookup table would map the BookID key to an electronic version of the text of the associated book. The recipe for the Bookshelf's fetchTheBook action might involve looking up the BookID, getting the associated e-text, and then providing it by email or some other transfer protocol to the patron. All of this could happen inside the computer, without involving any human beings at all.

[Footnote? Or teacher's guide? Or exercise: It is worth noting that the contract of the mail order/ Books on Tape and eBook versions of the fetchTheBook recipe are intertwined with the CirculationDesk's checkOut action in ways that the library patron's or library staff's implementations are not. This in fact represents a change of contract between the two implementations. As exercise: Have students act out the two versions, allowing the human user to be nasty and try to steal the book. Presume that the library has a gate that beeps if an un-checked-out book leaves the building; note that the book is already in the patron's hands and outside any beeping machine once fetchTheBook completes in both the mail order and eBook scenarios.]

2.3.4 Testing your Design

At this point, we have a rough design for the library checkout system. There are three major components: the CardCatalog, the CirculationDesk, and the Bookshelf. Each of these components has some behaviors (also called actions or services) as summarized in the table below. In addition, we have many BookIDs and PatronIDs, though -- for the present purposes -- these don't have any active behaviors.

Component Name	**Behaviors/Actions/Services**
CardCatalog	• lookup (a *keyword*) --> provides a *bookID*
	• checkOut (a *bookID* to a *patronID)* --> reports success or failure • checkIn (a *bookID*) --> reports success or failure

- verifyAvailability (of a *bookID*)
  --> reports availability

Bookshelf

- fetchTheBook (corresponding to *bookID*)
  -->provides the physical book

Using these components, we can build a system that handles the use cases of section @?@, above. For example, **Library-Lookup-Checkout** is a request to CardCatalog.lookup (yielding a *bookID*) followed by a CirculationDesk.checkout (of that *bookID* to the user's *patronID*). At this point, you should be able to flesh out a full playscript -- like the one at the end of "How do they interact?", but with all of the roles filled in -- for each use case. (If not, you have some more designing to do.)

These scripts -- together with the formal descriptions of each component and the recipes for each component's behaviors -- are the computer program that you are creating. Of course, there is the small matter of writing them in a language that the computer understands. But before you get to that step, it is a good idea to build your system out of human beings first. This gives you an opportunity to see how your system might work. It gives your customer an opportunity to decide that the specification needs to be changed. And it gives you an opportunity to design some tests that you'll want to use when you have actually built the computer program. Each of these is a valuable and important piece of software engineering.

This activity -- acting out the behavior of your system, testing your design -- is something that you can do by yourself, with a pencil and paper. But just as it is difficult to edit your own text, it is very hard to find the bugs in your own program when you are acting it out. It is much easier to see where things go wrong if you have a group of people each following your recipes very literally and precisely. So grab some friends and assign each one to a component. Give each component-actor a set of instructions -- a recipe -- for exactly how to perform each of his or her actions. Another actor plays the user and gets the use case scripts. No one is allowed to do anything unless the script specifies it; each actor has to follow the steps of the script literally. When information is transmitted from one component to another, write it down on an index card and let one actor hand it to the other. Before you begin, provide pieces of paper and pencils for any component actors who will need to remember things. Some of these may start with information already on their pages. For example, the card catalog actor will need to start with a lookup table listing the keywords and books in the library. The circulation desk actor will need another lookup table describing which patron currently has each book. (What should that list say when you begin?)

As you walk through this process, you will likely run into unexpected situations. For example, what happens if two check out requests come, one after the other, for the same book? Try this using the recipe for checkOut above. Do you see a problem? Can you fix the recipe? There is a problem; it as well as some possible solutions are described in the section on debugging, below. See if you can figure it out before you get there.

Your dramatization should make use of the test suite that you have been developing (and that you have continued to record in your engineer's notebook). You should make a point of running through each of the tests that you describe. It may be that several of your tests can be collapsed to make a simpler, more streamlined test suite. Or you may realize that you have neglected to test crucial features. By running through tests with human beings before you build your computer program, you can simultaneously identify problems in your design and build a more robust set of test cases for the computer program you will eventually write.

A dramatization also allows you an opportunity to show your design to prospective users. Ideally, your use cases capture everything that the user might want to do. Often, though, an opportunity to actually use the system -- or a mock-up of it -- helps the user to realize that additional features are necessary. Even if the specification doesn't change, it is still useful to hear your users' concerns at this early point in the software engineering process.

2.4 Building The Program

You have designed your program. You have identified use cases, decomposed the problem into its nouns and verbs, created descriptions of community members and recipes for their interactions, acted the whole thing out, and received approbation from your intended audience. You have, along the way, identified issues and revised your design to accommodate some of them; in other cases, you have listed areas for future revision and expansion of the system.

You are ready to begin developing your code.

2.4.1 Developing Code

It would be easy to imagine that designing a good program and trouble-shooting your design are sufficient preparation for constructing flawless software. However, just as a design needs trouble-shooting, an implementation will need debugging. You can make your debugging easier by implementing your system in pieces and stages, and by testing each one thoroughly as you go along. Designing your development process -- figuring out how to incrementally build your system so as to minimize the complexity of what's being tested at each stage -- is an important part of the development cycle. But even with the best design and development process, remember: *debugging is normal.*

Before you write any code, you should come up with a development plan. This plan is a sequence of steps that you will take to develop your program. At each step, you will write some code, test it, debug and revise it as necessary. Once the code for one step is working to your satisfaction, you should checkpoint that version of the code before moving on to the next step. Each step should include only a small amount of additional functionality. Your development plan should describe each of these steps, including what code you will write in each step as well as what new behavior you expect to see and what tests you will run to ensure that the new behavior -- as well as the old behavior -- works as expected. Notes that you have made up to this point about a test suite will prove particularly useful at this stage. Your development plan should be written in your engineer's notebook. As you proceed through it, you should verify your expectations, add comments whenever your code surprises you, and modify the plan to ensure that each step is small enough to be readily testable.

There are several things that you can do as you develop your code that will make your job easier. First, start simple. What is the most basic version of your program that you can imagine testing on its own? Start with the simplest, most stripped down functionality that you can think of. Or pick a component and simulate its interactions with the rest of the system rather than building the whole system at once. Add function incrementally. Always keep a working version (checkpoint!). Make minimal modifications, then test again.

In the library system, you might build a very simple masterList with just one bookID. Then, write enough of the CirculationDesk to be able to verifyAvailability of that bookID. (If the book is available, this action should always say so; if you manually set it to be unavailable, checkAvailability should reliably report that.) Or start with no books at all. (Now if the system tells you a book is available, you'll *know* you have a problem!) Once that is working, add a few more books and test their availability as before. Next, add one patronID -- make sure that that hasn't changed anything -- and then implement checkOut. Now you can check out a book, then verify its availability. What happens if you try to check it out twice in a row?

@@ *see exercise* # @@

Along with your development stages, think about how you will test each one. Set targets for what your code should be able to do at each point. What functionality can you demonstrate after you've built the most strip-down version? How would it respond to inappropriate input? Can you break it? When you add the next feature, how will you test it? Don't forget to test basic functionality after you add features; sometimes seemingly unrelated changes cause previously working aspects of your program to fail, especially if there are interactions in your design that you didn't yet discover.

Finally, don't forget to document your code as you write and test it. Explain what it does, how it works, and why you made these choices. Remember, the next person who reads your code may not know how it's supposed to work. Often, after a break, even you will have trouble remembering why you did what you did.

You should always implement your code in simple, testable stages, building on each stage only after it works robustly. Don't be afraid to move slowly and carefully through the software development process. Basic but elegant, well-tested, well-documented, and well-understood code will serve you better in the long run than featureful but poorly written/documented/tested code.

2.4.2 Compiling Code

In the previous chapter, we described the process by which a computation actually takes place. First, the instructions for that computation must be available. This is like having a script for the play. Second, the computer must execute those instructions. This is like having actors actually perform the play.

When you write a program, you go through a similar process. First you create the program, writing the script(s) that the computer will follow. Typically, you do this using a program called an editor. Later, you ask the computer to perform using those scripts. This is called running your program.

When you build programs using the programming language called Java (as we do in this book), there is an intermediate step that you must take. This is because Java -- as you write it -- is not directly executable by the computer. It is a bit like having written a play in English and then asking that it be performed in French. (Java is like English in this analogy; what your computer executes would be French.)

After you have written your program (and saved the file), but before you can run it, you must compile it. ***Compiling*** the program translates it from the version of Java that you write (and that is made to be read by people) into a different notation (called Java byte codes) that is directly executable by an appropriately equipped computer.

This point is important, if subtle. The Java program that you write is not directly executable by your computer. Instead, you must compile it, creating an executable set of instructions. Once it has been compiled, you can run this program as many times as you like. Compilation is a translation step that turns the Java you write into directly executable Java byte code. (Because the compiler starts with your Java program, that program is sometimes referred to as ***source code***: source for the compiler.) Compiling the program is not writing it -- you must write the program before it can be compiled -- and it is not running it -- the *result* of compiling the program is a computer-readable version of the script that can be run. You must write (edit, save) your program, compile it, and run it in order to see what happens.

Depending on the actual system that you are using to write your program, you may be more or less aware of when you switch from writing to compiling to running. Many programmers today use a special piece of software called an ***integrated development environment***, or ***IDE***. Typically, an IDE includes an editor, a compiler, a run-time environment (i.e., the ability to run your programs), and a debugger (on which more below). In a good IDE, you can move back and forth -- from one of these pieces to another -- easily. While this makes program development easier for skilled programmers, it can confuse beginners unless they keep in mind the differences between writing source code, compiling that source code, and running the resulting compiled program.

One of the nice (or not-so-nice) things about compiling code is that it gives you an opportunity to discover certain kinds of errors, or bugs, in your programs. For example, compilers can usually tell you when the code that you've written is not legal (Java). For example:

Figure 2.7. @@ draw pic of edited source code, pass through compiler, run in runtime environment (w/debugger)

- *6 +*4*
- *esle*
- {(})

These are called ***syntax errors***. Syntax errors can by typographic, like the transposition of the *s* and the *l* in *esle*, or they can result from mis-remembering a name (e.g., calling something *getSize* when it's really *getDimension*). A syntax error can also be the result of bad punctuation or of accidentally commenting out more (or less) than you intended.

Because syntax errors make your code illegal, the compiler will not be able to figure out what you mean and it will complain. Unfortunately, the compiler may not trip over the bug at the point where the syntax error actually arises. Often, the compiler will do its best to figure out what you mean and only discover that it is mistaken after it's done processing the line, the block of text, or even the whole file. So when a syntax error occurs, the compiler will tell you, but it may be difficult to figure out exactly what (or where) the syntax error occurs from the compiler's error message.

As you encounter compilation errors, keep track of what the error message is, where the compiler said the error occurred and, when you find it, what and where the actual error was. After a while, you will start to see patterns in how certain mistakes in the program cause the compiler to object. Usually, it's a good idea to start at the place where the compiler reports the error and work backwards, but this can vary tremendously from one compiler to the next. Learning to understand compilation errors is a good time to have someone around to ask questions of.

When you find a compilation error, you will need to go back and re-edit the source (Java) code file. Once you've eliminated the errors reported by the compiler, you will need to compile your file again. Sometimes, eliminating one compilation error will cause others to show up. Often, a compiler will only report the first few errors it finds. Eventually, though, you will eliminate all compilation errors, compile your code, and find that you can run it. Now, you can begin to test it.

[WARNING: Remember that you need to compile successfully each time that you modify the source code; otherwise, you could be running an old version of your program!]

2.4.3 Scaling Up Slowly

Hints for dealing with compilation errors:

- Check the line the compiler has identified as the sourc of the error.
- Check the line before the one the compiler identifies.
- If the compiler identifies a line containing a method invocation, check the method definition against the metho invocation.

Not all compilation errors are syntax errors.

At each step in your development plan, you will need to carefully test both new and old behavior of your code. In order to test your code thoroughly, you will want to draw on the test suite that you developed in specifying the problem -- including its use cases and guarantees -- and in designing your implementation and development plans. As you scale up the actual running software that you have built, you should continue to test basic functionality from earlier stages -- to ensure that it is still working -- along with the new functionality that you have added. It is OK to combine tests as you go along, but you should not generally drop a test entirely unless the same behavior is exercised by another test. It is important not to go on to the next step of your development plan until you understand what your code is doing at this step.

Be sure that you understand how your code actually behaves, rather than simply how you think it should behave. It is all too easy to kid yourself into believing that your code is correct. There is no better way to demonstrate your code correct than to thoroughly test it. Better, have someone else test your code for you.

Make certain that you know how your code will behave under inappropriate as well as appropriate circumstances. At all times, you should be able to describe what your code does as well as how it does it. If your code is surprising you, stop to figure out what it is doing and why. Surprises often come back to haunt you later if you don't take the time to figure them out when you first encounter them

Write documentation of your code as you go. Documentation is information that you leave for yourself or for other software engineers who may be unfamiliar with the code that you wrote. It should describe what your code does. It should not be an English version of the code; instead, it should summarize the functional behavior of the code: what job does it do? The documentation should articulate the formal and informal contracts that the code makes, including its assumptions and guarantees. Good documentation is so important that, throughout this book, we will include style sidebars that explicitly describe what kind of documentation is essential for each of a variety Java elements.

When you get a version of your software to work -- including testing it in every way that you can think of -- you will want to keep it around even as you go on to improve it. That way, you'll still have the working version when your next modification breaks it. Keeping a version is called ***checkpointing***. You should checkpoint your program whenever you have a working version and before every major revision, just in case it turns out to be a mistake. When you checkpoint, checkpoint the whole system, even if it is in multiple files. It can be hard to revert just one component of an interacting system.

Professional software developers and even advanced students often make use of versioning software to do their checkpointing. This software keeps track of different versions of your work and can help you compare these versions or even go back to an old one or merge two different sets of changes. Versioning software is a useful tool for serious software development (or for group projects, where different people may be working on different parts of the system at the same time).

You can do a simple sort of versioning yourself by periodically saving your project in a time-stamped backup. For example, this morning before you start working on your program, you can save a

Style Notes

Documentation

Each piece of code that you write should include text that documents (succinctly) what the code accomplishes. It should mention any assumptions made by your code. It should also tell yo why it is written the way that it is. Multi-line documentation shoul precede the code it documents.

There are certain things that your code itself says very well. For example, the pseudo-code for checking whether a book is available, below, makes use of good name choices and clear code flow to make much of its purpose clear:

```
to verifyAvailability of a bookID:

    1. masterList, please get bookID's record; call
       the associated PatronID whoHasIt
    2. report availability exactly if whoHasIt is
       LIBRARY_ID
```

Reading this code, the author's purpose should be pretty clear. Good coding practice is the first step towards maintainability

A nice piece of documentation for this code might paraphrase it, saying simply

```
/*
 * The following code reports
 * (to the requester) whether
 * the book is currently available
 * in the library.
 */
```

This documentation doesn't repeat what the code says, but does summarize how it behaves. In a longer, more complicated pie of code, the differences between the code and its accompanying documentation would be more apparent. A longer piece of code might also have additional information about its assumptions and guarantees. Examples illustrating these aspects appear in sections c this book where more complicated code is presented. Remember:

The most important function of documentation is to make it easier to understand and modify otherwise unknown code.

copy in a folder called 09-23-1030 (for 10:30 am on September 23[[Footnote: ...or on November 9th if you write your dates in the other order.]]). After you fix the bug that's been bothering you, you can save another copy in a folder called 09-23-1245, and when you're done programming for the day you can save that version in a folder called 09-23-2120 (9:20pm; programmers are often "night people"). The details of exactly how you do this will vary depending on the Java environment you're using, but the idea remains the same. If you think the version of your code you're currently working with is an improvement over a previous version, save a copy somewhere so that you can go back to it if your next change makes things worse.

2.5 Debugging is Normal

Debugging is normal. Everyone debugs. In fact, trial and error -- test and debug -- is a perfectly legitimate technique in building a piece of software. The important principle is that it should be *informed* trial and error; you should have a plan and a reason for trying the things that you are trying. (It's also a good idea to couple trial and error with a good checkpointing strategy; see above.) A good software engineer is sometimes a good experimentalist -- trying things out to learn from how they work and why they don't -- as well as a good experimental designer.

Leave room for debugging in your development process. Developing in pieces and stages so as to simplify your debugging process -- and designing in tests to verify behavior at each stage -- are important aspects of being a good software engineer. As you go, record your bugs -- the circumstances that reveal unexpected behavior, the sources of that behavior, and the solution to it -- in your engineer's notebook. Also record additional tests you might want to run later or concerns about other bugs that might arise down the line. These strategies will help prevent you from running into the same problems over again.

Debuggers can be your friends. Each particular development environment has a different debugger and it is important to understand how to work with yours. Consult your instructor or your documentation for specific help. Most debuggers let you stop your program at various points, look at objects or state, or walk through the program one step at a time. However, debuggers are not necessarily very good at working with programs where more than one thing can be going on at a time. You will need to have other tools as well.

Debugging, then, is like solving a mystery. Something is going on, and you need to find out what and why. You can approach debugging in most of the same ways that Sherlock Holmes would approach a mystery. You can sit and think. You can discuss the scenario with Dr. Watson (or anyone else who happens to be handy). You can play things out in your mind (or on paper, or with your friends). And, perhaps most importantly, you can set traps for the culprit: deliberately design experiments that will give you more information about what's going on, and where, and why, and under what circumstances this mystery arises. Your advantage (over Sherlock Holmes) is that your experiments don't run the risk of scaring off the culprit, so you can conduct as many of them as you'd like to try to solve the mystery. Just remember to save a copy of your code at the point that the bug arose, so that you can always go back to that version, rather than the one containing experiments and traps, to fix the bug.

2.5.1 Entomology Field Guide: A Taxonomy of Bugs

There are many different kinds of bugs that arise in programs and, as a result, many different ways to try to catch them.

First, there are syntax errors. Syntax errors are things that you write that are not legal Java. We have actually discussed these above, in the section on compilation, because a good compiler will catch your syntax errors. Although it can be tricky to learn to understand the errors your compiler reports, you will soon learn how to read them and find the underlying problem. Eventually, you'll probably even be grateful to your compiler for finding all of these bugs for you.

Not all mistyping leads to a syntax error. For example, if you replace a + with a - in your code, you will probably still have legal Java, but it is unlikely to do what you want. This is not, strictly speaking, a syntax error (and the compiler is extremely unlikely to think there's anything wrong it at all). It is a simple kind of ***logic error***. Logic errors occur when you write legal code to do the wrong thing. Generally, a logic error doesn't prevent the program from running; it just causes the program to behave strangely. Other simple logic errors include using a legitimate but incorrect name (e.g., calling verifyAvailability when you meant checkIn), or starting a counter off with the wrong value.

Simple logic errors are generally easiest to catch by adding steps to your program that print things out. For example, imagine that the library system keeps a count of how many books are currently in the library. Each time a patron returns a book, the program should add one to its counter, booksInHouse:

```
to checkIn a bookID:

    1. masterList, please put the pair bookID, LIBRARY_ID into the record
    2. add one to booksInHouse
    3. report success
```

But perhaps the program author used - instead of +, causing the number of books in the library to fall each time that a book is returned. Printing booksOut each time a checkout happens makes this error easier to find. This would mean adding an instruction (between #2 and #3) that says:

```
2.5 print the current value of booksInHouse
```

Actually, it would be a good idea to include where this line was printed and other things about its context. It is worth investing a little bit of time to print nice debugging messages, because you will often want to reuse them again later. A better version of this message might say:

2.5 print the following things:

- the phrase "In CirculationDesk checkIn, after returning book "
- the value of *bookID*
- the phrase "booksInHouse is "
- the current value of booksInHouse

A debugger may also let you watch the value of a quantity like booksInHouse directly.

Logic errors can also be a bit more complicated. For example, in the design section above, we asked what would happen to the code as designed for checkOut if two patrons tried to checkOut the same book, one after another. Imagine, for example, that someone named Abbott successfully checks out (the bookID corresponding to) a book called *Who's On First?* This means that the library's master list records the bookID for *Who's On First* as being associated with the patronID for Abbott. Five minutes later, Costello tries to check out the same bookID. Using the recipe for checkOut above, masterList is asked to put the information that *Who's On First*'s bookID is with Costello's patronID. So now Costello is recorded as having the book!

The problem that this reveals is a logic error. Even if each line of code that you wrote were, in itself, correct, your program would do the wrong thing. Acting it out is, in general, an excellent way to catch logic errors, but be careful that you (or your actors) do what the program *actually says*, not what you think it should say (or wish it would do). Sometimes, the logic error is too subtle for human actors to be able to recreate it. In that case, running the program using a debugger and inspector (or inserting a lot of printing statements that tell you what each part of the program thinks and knows as it happens) might be necessary.

In this case, the fix for the logic error is pretty straightforward. Costello should not have been allowed to check out the book, because the library didn't have it. [[Footnote: If the library did have the physical book, it would be because Abbott forgot to take it home, and it would be important to clear the book from Abbott's record before checking it out to Costello!]] So we need to add another piece to the checkOut recipe: First, the recipe should verify that the library is formally in possession of the book. A revised recipe might say

```
to checkOut a bookID to a patronID:

    1. masterList, please get bookID's record; call the associated PatronID whoHasIt
    2. report failure unless whoHasIt is LIBRARY_ID
    3. masterList, please put the pair bookID, patronID into the record
    4. report success
```

[[Footnote:

If you look carefully, you'll see that the first half of this new recipe is actually just a CirculationDesk's verifyAvailability recipe by another name. So, instead of writing it out explicitly here, we can use *this* CirculationDesk's verifyAvailability action directly:

to <u>checkOut</u> a *bookID* to a *patronID*:

1. (ask this CirculationDesk): please <u>verifyAvailability</u> for *bookID*; this responds with *bookID*'s availability
2. report failure unless it is available
3. masterList, please <u>put</u> the pair *bookID, patronID* into the record
4. report success

This technique -- reuse of your own recipes -- is excellent programming style whenever the recipe you're reusing has the same intent as the steps you're substituting it for. There are several reasons for this:

- Usually, the substituted steps are longer than the call to the other recipe, so this also helps keep you recipes shorter. (That's not true in this particular case, but "spread the jelly on the bread" is shorter than "repeat until the bread is full: pick up some jelly with your knife, put it on the slice of bread.")
- Even if it isn't textually shorter, recipe reuse can make your code easier to read. For example, saying "verify availability" tells you why you're bothering to get the patronID currently associated with the bookID.
- When you need to modify the code, there's only one place to do it. If the system were changed so that the library had two patronIDs -- ADULT_COLLECTION_ID and JUVENILE_COLLECTION_ID -- verifyAbility would be modified correspondingly. In this case, the version of checkOut that calls verifyAvailability would automatically work while the version that checks the masterList directly would break.

When, why, and how to reuse code in this was is covered in the chapter on procedural abstraction.

]]

This modification solves the sequential checkout problem. Recall, however, that we said that our library might have multiple checkout stations. What if Abbott and Costello each tried to check out the bookID for *Who's On First?* at the same time? This could lead to a problem even with the revised code: Each station might checkValidity, find the bookID in possession of LIBRARY_ID, and OK the transaction. Then both stations would go on to step 3, marking the book as checked out, each to a different PatronID. This kind of problem, which happens because multiple things are going on in the system at once, is called a ***<u>concurrency error</u>***. Concurrency errors and their solutions are the major subject of the chapter on synchronization. They can sometimes be identified by acting things out; at other times, it is more useful to have different pieces of your program report on which steps they are executing or about to execute and what they think is happening. It is always important to keep in mind what else might be going on in your program and whether one part of it can interfere with another.

Other errors arise, for example, when you misuse a piece of code. The CirculationDesk's verifyAvailability service is meant to check whether a particular book is available, not whether the CirculationDesk itself is available. Although that's not an error you're likely to make, similar misunderstandings of other code -- especially unfamiliar code libraries -- often lead to program bugs.

2.5.2 Good Tests Catch Bugs

Designing for debugging is a fine art. So is knowing which tests to run. The logic issue above -- Abbott and Costello each checking out the same book -- is a classic test; it is just the kind of misuse your system might not be designed to prevent. Two books with the same name (but different bookIDs) might be a good test for your system. A book that is lost -- never checked back in -- could present a problem. What would happen if the CardCatalog's lookup recipe returned a BookID that the CirculationDesk didn't know about? Could this ever happen in your system? What else could go wrong? Also think about the "normal" cases: each use case should have at least one corresponding test. As you become a more experienced programmer, you will build a mental catalog of various kinds of errors and the tests that catch them. Eventually, you will recognize and anticipate these kinds of issues. For now, your engineer's notebook is a good place to start this catalog.

Timing collisions -- two things happening at just the wrong times -- are another standard kind of problem for a system. Unfortunately, timing problems can be intermittent and so they are harder to identify or reliably replicate. Again, experience will give you better intuitions as to when a timing problem might be arising and how to prompt it to reveal itself. In the interim, debugging with a skilled assistant is useful, but in all cases make sure that *you* understand what was wrong with your program and that you fix it yourself.

Along with the catalog of tests and bugs that you will accumulate -- in your notebook and through your experience -- there are some techniques that can help you figure out what your code is doing -- right or wrong -- and help you to trap bugs.

2.5.2.1 Make state manifest

It is much easier to tell what your code is doing if you can see it work. A debugger may let you step through your program, one line at a time. A visualizer may even let you watch how things change. However, many debuggers and visualizers don't work very well with the kinds of programs that we will be discussing in this book, programs in which more than one thing can be going on at a time. In this case, you may have to get your code to tell you what it is doing without the help of the debugger or visualizer.

You can get your code to tell you what it is doing by adding steps to each recipe that print out (on the computer screen) what is happening. We did this with line 2.5 of the CirculationDesk's checkOut recipe. It can be a good idea to include the following steps in a recipe:

- When the recipe starts, a step that prints the name of the recipe and the message "starting".
- When any major change is taking place, a step that prints the name of the recipe and information about the change.
- When the recipe finishes, a step that prints the name of the recipe and the message "completing".

You can also include steps that print out where within a long recipe the instruction-follower is. Note that each printing step begins with the name of the recipe. If there are multiple objects that run the same recipes -- multiple CirculationDesks, for example -- it is important to include information as to which one is using that recipe step, too.

In addition to code that tells you which step in a recipe is happening, you may want to add code that can tell you about a particular object. For example, you might give each PatronID might have a recipe that prints any information associated with that patronID, such as the patronID number, the patron's name and address. [[Footnote: Java actually provides a particular recipe to do this -- it's called toString() -- but you need to supply the steps to make the toString() recipe useful. More on this in subsequent chapters....]] Or the CardCatalog might have a recipe to print the masterList, with all of its book circulation information. These kinds of recipes that let you see the state of things can be particularly helpful for debugging.

Even when you have fixed your bugs, don't be tempted to get rid of these recipes. You can leave them in place, but eliminate (or "comment out") the steps that invoke these recipes. After all, you never know when a new bug will pop up and you'll want to use these recipes again!

2.5.2.2 Explain it to someone.

One of the hardest things for a programmer to do is to recognize that the program is not doing what s/he thinks it should. Don't make the mistake of assuming that your logic is correct. Believe what the code is telling you. If you don't want to believe it, add more steps to get the program to tell you more about what is actually happening. Eventually, you will believe that it is doing what it says it is and, sooner or later, you'll understand why.

Sometimes, you can find a bug just by explaining how your code works. In fact, many bugs are caught by people spelling out why the behavior of the program is simply impossible....oops! The reason is actually quite simple: You know, better than anyone, how you wrote your program. You also know why you thought that the code that you wrote would do the right thing. But you probably didn't go over each detail all that carefully before you wrote the code and compiled it. When you actually take the time to explain, in detail, why you did what you did, you are likely to realize where your logic wasn't completely correct. Rumor has it that one college programming class has a requirement that a student seeking help with a program first explain the program's behavior to a teddy bear that they keep on hand for just such purposes. I'm sure that the teddy bear has found as many bugs as the lab assistants!

Imagine, for example, that your library program also keeps track of how many books are actually present in the library. But, somehow, the number of books in the library keeps getting smaller. In fact, after a while, it becomes negative! How could this be? Well, each time that the CirculationDesk runs its checkOut recipe, it reduces the number of books present in the library by one. And each time that it runs its checkIn recipe....There it is! Maybe you forgot to add one to the number of books in the library in the CirculationDesk checkIn recipe.

You'd be surprised at the number of bugs that are caught mid-way through an explanation. And if you can't explain how the program was supposed to work (in a clear, coherent, organized way) that's a telling sign, too. In fact, explaining BEFORE coding is a good idea to get your bugs out early. Remember: Developing software is an incremental process. It cycles back and forth, extending the behavior of your program and then verifying that it does what it should. Debugging is normal, and learning to debug your programs is an essential part of becoming a software developer.

Chapter Summary

- Clean coding and good documentation enhance the lifetime of your software.
- Program construction is a cycle of designing, building, testing, and then designing again.
-

Exercises

1. Write the **Library-Checkout** use case.
2. Add a **Library-Reserve-Book** use case , in which I ask the library to hold a specific book for me, and a **Library-Lookup-Reserve** use case, which combines **Library-Lookup-Can't-Checkout** and **Library-Reserve-Book**. How about **Library-Checkout-Reserve** (check out a previously reserved book), which ought to look a lot like **Library-Checkout** but include a step to verify that the checkout patron is the person who held the reservation? Also include tests for these use cases.
3. Write down the scripts (as described at the end of "How do they interact?") for each of the use cases listed in this chapter:
    a. **Library-Lookup**
    b. **Library-Checkout**
    c. **Library-Lookup-Checkout**
    d. **Library-Lookup-Can't-Checkout**
    e. **Library-Checkin**
    f. *** Library-Reserve-Book**

   g. *** Library-Lookup-Reserve**
   h. *** Library-Checkout-Reserve**

4. Design a book reservation system that allows a patron to request that a particular book be held for his/her future checkout.
   a. Which component of the library should be responsible for recording that a book is on reserve?
   b. Write an action contract and the recipe for placing a book on reserve.
   c. Incorporate that recipe into a script for for Library-Reserve-Book
   d. Describe at least one test that you could use to verify the behavior of this script
   e. Write an action contract and the recipe for checking out a previously reserved book.
   f. Incorporate that recipe into a script for Library-Checkout-Reserve
   g. * How should the reserve system affect the existing checkout code? What could go wrong happen if you did not modify the existing checkout code? How could you fix it?
5. Add three more steps to the library development plan described in the "Developing Code" section of this chapter. Be sure to include a test sequence for each.
6. Use the original checkOut recipe. What happens if two check out requests come, one after the other, for the same book?
7. If you've edited (saved), compiled, and run your program and want to run it again, do you need to compile it first?
8. If you've edited (saved), compiled, and run your program and you modify (edit, save) your program, what do you need to do to see what the program does now?
9. * The contracts of the mail order/Books on Tape and eBook versions of the fetchTheBook recipe are intertwined with the CirculationDesk's checkOut action in ways that the library patron's or library staff's implementations are not. This in fact represents a change of contract between the two implementations. Have students act out the two versions, allowing the human user to be nasty and try to steal the book. Presume that the library has a gate that beeps if an un-checked-out book leaves the building; note that the book is already

in the patron's hands and outside any beeping machine once fetchTheBook completes in both the mail order and eBook scenarios.

10. Extend the library system described in this chapter to provide the following additional features of a library reserve system:

    a. Notification when a reserved book becomes available.

    b. Allowing a second person to reserve the book even before the first reservation is filled.

    c. Limiting the number of reservations that a single patron can make.

11. Extend the library system described in this chapter to provide the following additional features:

    a. Datestamp checkOuts

    b. Track when books become overdue

    c. Send overdue notices to patrons whose books are outstanding

    d. Charge a fine when an overdue book is checked in

    e. Refuse checkouts to a patron who has accumulated more than a specified amount in fines

12. Design an electronic calendar system with an alarm notification for each event. You may assume that you have access to a clock program library. (You may make further assumptions about the behavior of the clock library, but you must spell these out explicitly. Alternately, you may use the clock found in Chapter [@@animacies].

13. Design a restaurant system including at least 3 constituent community members. Write scripts for each of their behaviors.

Computers and Cognition Laboratory of the Franklin W. Olin College of Engineering and formerly at the MIT AI Lab and the Department of Electrical Engineering and Computer Science at the Massachusetts Institute of Technology.

Questions or comments:
<webmaster@cs101.org>