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

Introduction to Interactive Programming

By Lynn Andrea Stein

A Rethinking CS101 Project

Chapter 1.
Introduction to Program Design

Chapter Overview

- What is a computer program?
- What are the parts of a program? How are they put together?
- What kinds of questions does a program designer ask?

In this chapter you will learn how a computer can be controlled by a set of instructions called a ***program***. This chapter introduces two different aspects of computation: single-minded instruction following and coordination among instruction followers. The programs in this book involve both aspects of computation.

The first aspect of computation is step-by-step instruction following, like the process of making a sandwich. This kind of computation is a sequence of instructions that produces some desired result. The question that drives this part is "What do I do next?" Pieces are put together using "Next,...", "If ... then ... else ...", and "until...". This kind of computation has an end goal to be accomplished by execution of these instructions. The programs in this book use short sequences of instructions, executed over and over, to create entities that can provide services or respond to requests (e.g., a sandwich-maker).

The second aspect of computation involves coordinating among many of these instruction-following entities. This is like gathering sandwich-makers (and table-waiters and others) together to run a restaurant. This kind of computation is creating (and managing) a community. The driving questions are "Who are the members of the community?", "How do they interact?", and "What is each one made of?" The members of the community -- the instruction-following entities -- are glued together through their interactions and communications. Executing this kind of computation provides an ongoing program such as your car's cruise control, a web browser, or a library's card catalog.

When you finish this chapter, you will know the basic questions to ask about every computational system. These questions will allow you to begin to design a wide variety of computer programs.

Objectives of the Chapter

1. @@@
2. To understand what programs are made of.
3. To be able to recognize and articulate use cases.
4. To be able to recognize and articulate requirements, assumptions, and (desired) guarantees.
5. To be able to read and construct sequential, conditional, and looping programs using English steps.
6. To be able to design communities of interacting entities using English specifications.
7. To appreciate the difference between a program and its execution.

1.1 Computers and Programs

Computers provide services. A suitably equipped computer can retrieve a web page, locate the book whose author you're thinking of, fly an airplane, cook dinner, or send a message to your friend half way around the world. In order for a computer to do any one of these things, two things must happen. First, the computer must be told *how* to provide the required services. Second, the computer must be asked to do so.

The how-to instructions that enable computers to provide services are called ***programs***. A computer program is simply a set of instructions in a language that a computer can (be made to) follow. When the computer actually follows the program instructions, we say that it is ***executing*** that program. The program is like the script for a play. It contains instructions for how the play should go. But the script itself is just a piece of paper: no actors, no costumes, no set, no action. Executing a program is like performing the play. Now there is something to watch.

This analogy goes further, too. The same script can be performed multiple times, just as the same program can be executed again and again. If audience reaction (or the director's interpretation, or the theater, or the time of day) influences the performance, two performances of the same script may be quite different. Similarly, user input, hardware, software, or other environmental circumstances may make two different executions of the same program quite different from one another. (Think of running the same word processing program on two different occasions; the experiences are extremely different even though the computer follows the same general-purpose instructions both times.)

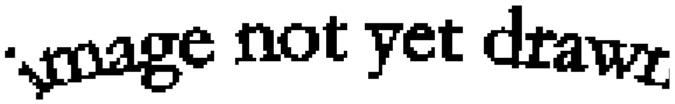

Figure 1.1. Execution of a program.

Figure 1.2. Performance of a play.

When you sit down at a computer, someone else has already told it how to do a lot of things. For example, when you press the power switch, it ***boots*** up -- gets started running -- in the way that it has been instructed to. Personal computers typically come with a fairly sophisticated set of startup instructions already installed. Simply turning on the computer causes the computer to execute this startup program. [[Footnote: Starting a computer is called "booting it up", presumably from the phrase "pulling yourself up by your bootstraps". The startup program that a computer executes each time that it is turned on is called the computer's "boot sequence".]] Each computer has a program that it runs automatically. The program that your desktop or laptop PC runs is called its operating system. A disk drive -- which is really a separate computer plus the electronic equivalent of a huge filing cabinet -- comes equipped with instructions for how to retrieve information from (or store information in) that filing cabinet plus how to transmit that information across the cable that connects the disk drive with your "main" computer. A microwave oven comes with a computer that follows instructions for how to tell time and how to turn on its microwave generator for specified periods. The library's card catalog provides lookup services. Your car's cruise control accelerates and decelerates to keep your car moving at a steady rate. A web browser fetches and displays information it retrieves from the hard drives (file cabinets) of computers scattered around the world at your request (with the assistance of the "web server" programs running on those distant computers as well as the network (transmission) services provided by a set of intervening computers.

Figure 1.3. Two performances of the same play. The experience may be different, but there will be similarities as well.

When you load a new piece of software onto your computer -- a cool new game, for example -- what you are actually doing is giving your computer a copy of the program -- the set of instructions that tells it how to do display graphics and make appropriate sound effects or whatever it is that the particular piece of software does. Writing down these instructions was the job of the person (or people) who wrote the software, the ***programmer***. Loading the software makes the instructions (the script) available to your computer. Just having these instructions lying around doesn't do you much good, though. To actually play the game (perform the play), you need to do one more thing. You need to run the program. [[Footnote: Some computer games can be run off of removable media, like CD ROMs. In this case, you don't need to load the program onto the computer, but you do need to make sure that the disk is in the drive, i.e., that the instructions are available to the computer. Also, some programs run automatically when loaded, e.g., applications for browser plugins.]] Tomorrow, if you want to play the game again, you only have to run it; you don't have to start by loading it onto your computer.

1.2 Thinking like a programmer

A computer program -- "how-to" instructions for your computer -- must be written in a language that the computer can follow. There are many languages designed for instructing computers. These languages are called ***programming languages***, and they are typically quite different from the kinds of languages in which people talk to one another. One of the main differences between talking to a person and programming a computer is the increased level of precision required to tell a computer how to do things. With people, it is often possible to give very vague instructions and still get the behavior you want. A computer has no common sense. You must be very specific with it. Your instructions must be step by step, in great detail. We've already noted that a program -- these instructions -- are like the script of a play. Now, we'll look at an individual's instructions -- one role -- as the kind of step by step instructions you might find in a recipe. In some ways, programming a computer can be a lot like talking to a very young child or a creature from a different planet.

Imagine teaching a Martian how to make a ***peanut butter and jelly sandwich***. You need to give detailed, step by step instructions:

1. Get a loaf of bread.
2. Remove two slices of bread and put them on the counter.
3. Get a jar of peanut butter. Put it on the counter, too.
4. Get a jar of jelly. Put it next to the peanut butter.
5. Get a knife.
6. Open the jar of peanut butter.
7. Pick up a slice of bread.
8. Using the knife, pick up a glop of peanut butter and spread it on the top of the slice of bread.
9. ....

These instructions tell the Martian, in very specific terms, what to do. To follow the instructions, the Martian simply needs to perform each step, one by one, in the order given. As long as each of these instructions is one that the Martian knows how to

Figure 1.4. Making a peanut butter and jelly sandwich.

perform, when the Martian finishes executing this program, the Martian will have a peanut butter and jelly sandwich.

If there is an instruction here that the Martian does not understand, that instruction needs to be rewritten in more detail so that the Martian will be able to execute it. For example, "pick up a glop of peanut butter" might require further explanation:

a. Insert the knife blade half-way into the jar of peanut butter.
2. Remove the knife from the jar of peanut butter at a slight angle so that some peanut butter is carried out of the jar by the knife.
3. ....

An instruction that needs further explanation before the Martian (or computer) can execute it is one that we call ***high level***. Java -- the programming language used in this book -- is a high level language. Detailed instructions that tell your computer how to execute Java statements are supplied as a part of your Java ***programming environment***. You can build your own (even higher level) instructions in Java. In this case, you'll need to explain them to the computer (in Java). In general, we can use high level steps in our programs only if we can supply additional instructions to explain how to actually execute these higher level steps.

Although we don't know what instructions Martians are likely to understand, a programmer knows what kinds of instructions are a part of the particular programming language in which s/he is developing a computer program. In this book, we will use a programming language called Java. As you read this book, you will learn how to think like a programmer and how to write instructions that computers can understand. You will also learn specifically about the kinds of instructions that are part of the Java programming language.

As a programmer, you will design sequences of steps much like the peanut butter and jelly sandwich instructions. The goal of such a sequence is to get something done, to find an answer or to create something. In order to design a program like this, you will need to repeatedly answer the question, "What do I do next?" until you have reached your desired result. In many ways, this approach makes computers seem much like sophisticated calculators. In fact, this is where computers got their start: the word "computer" used to refer to people who did (mathematical) computations, and the original mechanical computers were designed to perform these computations automatically.

When you are designing a program, you should ask yourself, "What do I do next?" You don't necessarily have to write out all of the basic steps in one long sequence. You can group them together in bigger, more abstract, higher level chunks:

I. Assemble the ingredients.
2. Spread the peanut butter.
3. Spread the jelly.

4. Put the sandwich together.
5. Clean up.

This is a perfectly good set of instructions. But, as in the case of the Martian who didn't know how to "pick up a glop of peanut butter", these instructions will require further elaboration.

This way of thinking about programs -- outlining the big pieces, then breaking each one down further -- is called ***top down***. A programming language such as Java allows you to make up your own high level steps, like "Assemble the ingredients", and then to explain how to do this: "1. Get a loaf of bread...." Your program is complete only when every line is either understandable by the computer or further explained in terms that are understandable by the computer. When you are done asking yourself "What do I do next?" you must then ask "How do I do each of these things?" until every line of your program is something that the computer knows how to do.

1.3 Programming Primitives, Briefly

What kinds of things do computers know how to do? Most computers don't know how to make peanut butter and jelly sandwiches. Most computers do know how to manipulate numbers and also other kinds of information, like words. In the Java programming language, you will find tools that let you send messages to other computers on a network or create windows and buttons to communicate with people using your programs. Other computers may have special kinds of instructions. A robot control system has instructions that tell the robot when, where, and how to move. A security system may have an instruction to sound an alarm. These are the basic instructions out of which programs for each of these systems can be constructed.

These basic instructions can be combined by ***sequencing*** them, as we've already seen. They can also be grouped into mini-programs and given names, like "Assemble the ingredients". These names can then be used as new instructions. When the computer needs to execute one of these new instructions, it simply looks up the rule for how to do it. (When the Martian needs to assemble the ingredients, it uses the detailed instructions that begin "1. Get a loaf of bread....")

Figure 1.5. A sequential script.

Instructions can also be combined in other ways. Sometimes, there is a choice to be made. For example, after spreading a glop of peanut butter on top of the bread (step 8), the next step in the peanut butter and jelly program might say:

9. If the top of the slice of bread is covered in peanut butter, go on to step 10. Otherwise, go back to step 8.

This step contains a choice; the next step might be 8 or it might be 10, depending on whether the slice of bread is full. The Martian (or computer) executing this program will have to keep track of which step comes next. This kind of choice step is called a ***conditional***, and it is a common construct in programming languages. It is especially useful when the answer to the question "What do I do next?" depends on something you won't be able to figure out until you're executing the program.

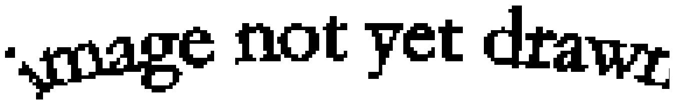

Figure 1.6. A conditional script.

We might want to go further, replacing steps 8 and 9 with a new kind of step that says

8. Repeat the following sub-steps until the top of the slice of bread is completely covered in peanut butter
    a. pick up a glop of peanut butter
    2. spread it on the top of the slice of bread.

This step ("repeat until") is called a ***loop***. It, too, is a common construct in programming languages. Some loops tell you to keep going until something is true (like the bread becoming full), while others tell you how many times to do the steps inside the loop. Some loops even go on forever. For example, a clock is basically a loop that moves its hand(s) (or changes its display) once a minute. Loops are especially useful when part of "What do I do next?" is to repeat (almost) the same thing several times.

Figure 1.7. A looping script.

Sometimes, a collection of steps is so useful that we want to give it a name and treat it like a basic step. For example, the loop that we just wrote -- repeat {pick up a glop, spread the glop} until the bread is covered -- is a nice summary of

II. Spread the peanut butter.

When we originally made that list of higher level instructions above, we had to further specify how to do each instruction. Identifying the high level pieces first is top-down design. Now, we're looking at the same problem from the other side. We have identified a set of useful instructions that accomplish the spreading. We want to give this a name (e.g., "spread") and be able to use it again. For example, we might want to use it again to spread the jelly. Identifying the low level pieces and grouping them together is called ***bottom-up design***. Giving a name to the group of steps so that we can use it again is a particular kind of bottom-up programming called ***procedural abstraction***.

Each of the techniques described above -- sequencing steps, conditionals, loops, and procedural abstraction -- is an important part of building computer programs. You will learn more about how to do these things in Part 2 of this book. These are the pieces that a programmer uses to answer the questions "What do I do next?" and "How do I do each of these things?" But this is only one part of the programming problem. If a program is the script of a play, we have so far been talking about a single actor in a single role, a one-character monologue. Real programs often involve many separate roles. The complete script involves not only each individual player's lines and actions, but also stage directions specifying how the different roles interact. The second part of programming is coordinating the activities of many interdependent participants in a computational community.

1.4 Ongoing Computational Activity

Some computer programs are very much like peanut butter and jelly sandwich making instructions. They start with some ingredients and -- step by step -- calculate whatever it is they're designed to create, producing an answer or result before stopping. The original mechanical computers, which mimicked human computers performing mathematical calculations, were very much like this. Sometimes, you would bring your program to a computer operator and then come back the next day for the result!

Today, most computer programs aren't like this. Instead, computer programs today are constantly ***interacting***. They may interact with people, machines, other computers, or other programs on the same computer. For example, a word processing program or spreadsheet waits for you to type at it, then rearranges things on the page or recalculates values as you type. A video game moves things around on your screen, some in response to you and others by itself. A web browser responds to your requests, but also talks to computers all across the network. The cruise control system for your car responds to road conditions, sensor readings, and your input. A robot control system interacts with the robot and, through the robot, with the robot's environment, perhaps with no human input at all.

These computations aren't concerned with solving some pre-specified problem and then stopping. Most computations of interest these days are things called servers or agents or even just applications. Most of them have some basic control loop -- a core sequence of instructions to be executed over and over again -- that responds to requests or other incoming information continually. These computations are ***embedded*** in an environment and they ***interact*** with that environment: users, networks or other communication devices, physical devices (like the car), and other software that runs at the same time.

Not only do programs interact with things around them, a program may also have interactions inside of it. In fact, each of these programs may itself be composed of many separate pieces that interact with each other (as well as with the world outside the program). Coordinating the activity among the many entities that make up your program -- and their interactions with the world around them -- is the second aspect of computer programming.

This is kind of like taking a group of Martians and organizing them to run a restaurant. Some of the Martians will take orders from and serve food to the

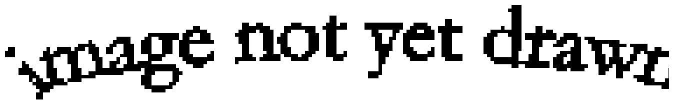

Figure 1.8. Participants in the community (chef, waiter).

customers. Other Martians will need to cook food for the customers. Still others will need to check on supplies, make change, or coordinate other aspects of the restaurant's operation. Each of these Martians will provide services to and make request of other Martians (or to the restaurant's customers or suppliers or other parts of the environment in which the restaurant is embedded). Coordinating the interactions among these Martians (and between the Martian restaurant and its environment) involves different kinds of questions from the instruction-following "What do I do next?"

Before we turn to the coordination of activity, though, let's look closely for a moment at one of the Martians who will staff our restaurant. We will see that, deep down, "peanut butter and jelly" programming still has an important role to play in creating computational activity. Keep in mind that this Martian represents just one of the many things going on in our restaurant.

The instructions that a Martian chef follows might look very much like this:

1. Pick up a new food order.
2. Find the instructions for the dish ordered and follow them.
3. Put the completed dish and the order information on the counter for pickup.
4. Go back to step 1.

Step 2 of this program is the kind of "higher level" step that we described above. It is not itself complete; instead, it refers to other, more detailed instructions to be followed. For example, if an order comes in for a peanut butter and jelly sandwich, the Martian chef will need to use the instructions developed in section 1.2 for how to make a peanut butter and jelly sandwich. Like a computer, the Martian is following simple sequenced steps written in a language that it understands. But while this Martian is making a peanut butter and jelly sandwich, another Martian is asking the customer at table 3 whether she would like some more water. Later, the Martian waiter will come into the kitchen and pick up the sandwich that the Martian chef just made. And when the Martian chef is done making the peanut butter and jelly sandwich, that Martian will turn to the next food order, continuing its ongoing interaction.

Figure 1.9. Echo script ends with goto step 1.

The peanut butter and jelly style of program instructions is an important part of how the Martian chef does its job. But the Martian chef's instructions are not simply the steps of the peanut butter and jelly program. The basic structure of the Martian chef program is an ***infinite loop*** -- a loop that goes on forever. This program accepts requests (in the form of new food orders) and provides services (in the form of the completed dishes) over and over again. We sometimes call this kind of loop -- one that provides the main behavior for a participant in the interactive program community -- a ***control loop***. Many program community participants take this form, and we will look more closely at control loops in Part 3 of this book. Programs with ongoing central control loops like this are the members of our interactive computational community.

1.5 Coordinating a Computational Community

At its most basic level, every computer program is made of instructions that are followed, one by one. But a single computer program may have many instruction-followers inside it, just as our restaurant is run by many individual Martians. When you look at the whole program -- like the whole restaurant -- you don't necessarily see the individual instruction steps. Instead, you see coordinated activity among a group of interacting entities. The behavior of this community -- e.g., providing customers with hot meals -- is not the responsibility of any particular member of the community. Instead, it is the result of many community members working together in a coordinated fashion.

Building modern interactive software involves something very much like organizational design. We call this part of programming "constituting a community of interacting entities". The programmer's job to figure out how to tell the computer what to do, and no matter what the specific problem to be solved may be, there are fundamental questions that each programmer must ask. Designing a computation which is a community of interacting entities involves figuring out who the members of this community are, how each one works, and how they interact. This is like setting the cast of a play, or deciding what the sub-units of your business will be, as well as how they should interrelate. In planning the organizational structure of your business (or program), you also have to figure out how each unit works and what -- and how -- they are supposed to communicate. These are the big questions of this second aspect of programming.

When you are designing this kind of activity, you ask yourself several questions:

- What is the desired behavior of the program?
- Who are the entities who interact to produce this behavior?
- How does each one work?
- How do these entities interact?

In the remainder of this section, we will expand these questions and begin to explore them in somewhat greater detail. Understanding these questions and their ramifications is the theme of this entire book. Coordinating communities is a special focus of Part 4.

1.5.1 What is the desired behavior of the program?

Before you can design a system to solve your problem, you must know what your problem is. This involves knowing not only what you want, but how it should work or fail to work under a variety of different circumstances.

One way to determine what your program needs to do is to envision how it will look to someone who interacts with it. A person who interacts with a computational system is called a ***user*** of that computational system. A description of a particular interaction between a user and a computational system is called a ***use case***. [[Footnote: One computational system can also be thought of as a user of another, and one computational system's experience interacting with another is a sort of use case as well.]] A good use case includes

1. the prerequisites: what must be true for the use case to arise
2. the set of possible actions and interactions, perhaps described in terms of conditionals and loops
3. the effects: what changes are made to the computational system and to the user as a result

For example, a typical use case for a restaurant might be: walk in, sit down, get the menu, order food, be served, pay, leave. The (usual) prerequisites are that the customer is hungry and has money. The postconditions are that the user is less hungry and has less money, while the restaurant has more money and less food (not to mention some dirty dishes). Another use case for a restaurant might be: walk in, find out that the wait for a table is too long, leave. In this second use case, no money changes hands.

Use cases are a good way to identify the behavior and interactions that you expect from a system. Use cases can be specified with varying degrees of formality. In this book, we use a relatively casual format for specifying use cases. For a more rigorous approach, see @@.

Some questions that you ought to be able to answer about your desired program include:

- What services should your program provide?
- What guarantees does your program make about these services?
- Under what assumptions (circumstances, conditions) does your program make these guarantees?

What can we say about the behavior of the restaurant? In answering this question, we consider both the experiences of individual customers -- reflected in use cases -- and the ongoing properties that the restaurant must maintain, such as remaining solvent. A basic specification of the service provided by the restaurant might be: Each customer is seated at a clean table, the order is taken, food is served, a bill presented, and payment collected.

There are a number of guarantees we want to make about these services. For example, customers should not have to wait for an unduly long time. Different parts of the restaurant must communicate; customers should not be charged for food that they were not served, etc. Over time, the restaurant should take in at least enough revenue to cover its operating expense. Supplies should not run out, nor should they rot.

We will make certain assumptions in order to be able to provide these guarantees. For example, the "timely service" guarantee will only be possible if the customer load on the restaurant is reasonable. We might decide that we will only be able to uphold this guarantee if the number of people wanting to eat in the restaurant at one time never exceeds its capacity, and if the rate of arrival of these people doesn't exceed the rate at which the restaurant can serve them. [[Footnote: How many customers the restaurant can handle at the same time is called its bandwidth. How quickly each one can be served is called its latency. The number of customers per hour that the restaurant can handle is its throughput. These quantities -- bandwidth, latency, throughput -- are common measures of program performance.]] These assumptions should be made explicit, and we will also need to say what happens when they are violated. (In this case, the timely service guarantee won't be upheld, but how slow the service gets should be related to how overloaded the restaurant is.)

There are other assumptions we do not make about our program, and we can articulate these as well. We do not assume that only one customer will be served at a time. Instead, we expect that multiple tables must be handled (roughly) simultaneously. It certainly won't do to wait until the first customer has eaten, paid, and left before addressing the second. We also permit different interactions with each table to be handled simultaneously or at least overlapped; food may be cooking while checks are being written up.

This description is still fairly general, and we can imagine making it more specific. (For example, are customers constrained to ordering off of a menu?) In general, the more detail you can give of what your program ought to do, the easier your task will be in designing and building it.

1.6 Designing the community's members

The previous section was concerned with understanding the desired behavior of our program as a whole. In this section, we look at how the community is built. As with single recipes, sometimes whole programs are designed top down, first figuring out what the program should do and then breaking it into separate interacting roles. At other times, you will begin with pieces -- bottom up -- and combine them to create a community that accomplishes what you need. But whether you start at the top or at the bottom or do a bit of each, you will eventually need to answer the question of *who are the members of the community?*

This question can't be answered in isolation, because each and every decision you make about *who* the entities are is also at least a partial commitment to *what* they are and *how* they work. So answering this question is in many ways like solving the whole problem. The trick is to answer this question in fairly high-level, general terms, then to sit down and try to figure out the answers to all of the *what* and *how* questions. In answering those, you'll almost certainly have to return to this question and rearrange your answer a few times. This is fine; it's even typical enough to have a name: ***incremental program design***.

How do we know how to divide behavior? Often, we do so by enumerating coherent sets of resources and grouping them. The kitchen -- the stove, the food, the pots and pans and plates -- provides the impetus for one community role. The customer -- getting orders, delivering food, etc. -- motivates another. But there is more than one way to divide responsibility. A good division is one that makes the next question -- *how do the community members interact*? -- easier to answer.

In the restaurant, an appropriate high level division of labor might have a wait staff unit (the people who deal directly with the customers), a kitchen staff unit (the people who cook the food), and a financial unit (who keep track of how much which things cost, collect money, and buy supplies). At this point, we haven't committed to whether these are three roles played by a single Martian, three separate Martians, or even three groups of several Martians each. The final question we'll need to ask is *what is each one made of?*: How is each member of the community built?

1.6.1 How do the community members interact?

This question concerns coordination and communication among two or more entities. In many ways, this question is just our original question -- *what is the desired behavior of the program?* -- all over again. Program behavior was determined by looking at interactions between a program and the environment in which it is embedded. Now, we are asking about the behavior of a part of the community. Use cases can be a good tool to help understand within-community interactions as well.

Each entity in your program is a miniature system unto itself. Some of the questions that you should ask about how these entities interact include:

- What are the entities' interfaces?
  - What promises does each one make?
  - What contracts does it fulfill?
  - What services does it provide?
- How do they communicate?
  - What mechanisms do they use?
  - What interaction patterns do they use?
  - How do they preserve liveness, i.e., make sure that things keep moving?
- What interaction patterns are possible?
- What happens when something goes wrong?

Interactions are often described in terms of protocols. A ***protocol*** is the specification for an interaction between two entities. For example, a common protocol for the interaction between the wait staff and kitchen staff of a restaurant involves a slip of paper with the customer's order written on it. The waiter hangs this piece of paper in the window over the kitchen's food pickup counter, a place where it will be easy to find when someone from the kitchen is ready for a new job. When a member of the kitchen staff is ready to process the order, the piece of paper is removed and used to guide the food preparation. When the order is ready, it is placed on the food pickup counter together with the original order slip. This identifies the food with the original request when the waiter returns to retrieve it. The slip of paper serves as a crucial reminder of several associated pieces of information: what was ordered, by whom, and where they are seated.

This is a ***data structure based protocol***; it involves the use of a separate piece of information to keep track of what is happening with each order. The state of that piece of information -- where the paper order slip is -- tells the kitchen and wait staff what they need to know about how far along the order has gotten and whether it's ready. Contrast this with a simple ***request/response protocol*** in which the waiter stood at the entrance to the kitchen until from when the order was placed to when it was completed. In this protocol, the waiter can't do anything else until the kitchen has responded to the order request (by producing the cooked dish). Obviously, these two protocols will lead to restaurants that run very differently from one another.

Protocols can also address temporal issues. For example, the wait staff/kitchen staff interaction described in the preceding paragraph needs to happen in ***real time***, meaning that the protocol itself can't introduce significant delays. There must also be guarantees made about the frequency with which the wait staff checks for completed dishes (or the kitchen staff for incoming orders). If assumptions such as these are built into protocols, they must be documented so that they are maintained in the behavior of participant entities.

Not all interactions are real time. For example, the wait staff interacts with the financial unit by obtaining prices for food and turning over any moneys collected. These interactions could happen in ***batch***, meaning that it is OK for the wait staff to get the price list at the beginning of the week or for money to be handed over at the end of the day. [[Footnote: Batch processing is like the old-fashioned computations in which you handed your program to a computer operator and came back the next day for your results.]] The difference between real time and batch interactions is only one dimension that must be determined in order to coordinate the activities of the members of your computational community.

A protocol specifies the ***interface***, or meeting, between various entities in the community that constitutes your program. These interfaces are often captured in contracts that spell out the behavior that will be provided. Once the interfaces have been thoroughly fleshed out, each entity can in theory be implemented by a separate programmer (or team of programmers) provided that it is built to spec, i.e., that it meets the specifications of the agreed-upon interface.

1.6.2 What is each member of the community made of?

Finally, we look inside the members of our community. When we do so, we may find that a particular community member is itself a community of interacting entities. Or we may find that it is a relatively simple entity, performing a single task over and over again. Just as in sequential instruction-following, the most basic building blocks are combined to build higher level components, which can then be combined again and again to build increasingly more complex systems.

At the top level, before we could talk about who the community members were, we needed to know what the community would be doing. Similarly, before we can build a single constituent entity, we need to know how it's supposed to fit into the community. So answering the question of *what goes inside?* requires knowing something about *how they interact*. Before you can design the structures that go inside, you need to be able to specify what the community member will do. After all, specifying what interactions each entity needs to support goes a far way towards telling you whether whatever goes inside meets the requirements of the community.

Some subsidiary questions to ask about how each of the entities is constituted include:

- What responsibilities does it have?
- What guarantees (promises, commitments) does it make? Under what assumptions?
- What resources does it control?
- How does it work?
- Is it a community, too?

What responsibilities does it have? The restaurant's wait staff might be responsible for greeting the customers in a timely fashion, supplying each one with a menu (a structure that the program will have to provide and keep updated!), taking the order, delivering it to the kitchen staff, picking up and serving the cooked meal, obtaining a price from the accounting entity, and obtaining payment for that amount from the customer.

What guarantees does it make? The wait staff might guarantee to communicate with (most of) the customers within minutes, provided the total number of customers is limited and the maximum time spent with each is under a certain amount. It might also promise to deliver food within some small amount of time after it's done cooking, provided that the kitchen staff notifies the wait staff in a timely manner.

What resources does it control? The wait staff controls menus, knows which food items were ordered by which customers, and is the only part of the restaurant that deals directly with the customers. And so on.

When it comes to *how does it work?*, there are two kinds of answers. One answer is that the behavior of the entity is accomplished by a single rule-follower running an interactive control loop. We saw an example of this when we considered the Martian chef earlier. In this case, we ask "What does the Martian do next?" over and over, until we wind up with a well-defined set of instructions for this Martian to follow.

Figure 1.10. Chef control loop vs. complex community (2 pics).

The other possible answer to the question *how does this entity work?* is that this entity is itself a community. The wait staff might be further divided into the person who takes the order, the person who clears the table, and the person who serves the wine. In this case, we need to figure out how to build each of these entities, asking again *what goes inside each one?* The problem of figuring out how to coordinate the activity of a community continues until each community member is a single (rule-follower) Martian. Then we ask about the instructions this Martian follows, using the vocabulary of section @@1.3.

In practice, the task of implementing an entity to match a given specification often results in questions about or revision of that interface. Programming is not so neat a task as students of computer science would often like to believe; there's a cycle of specification and implementation, debugging and testing, usage and revision, that characterizes almost all real-world software. The later stages of this process are sometimes called the ***software life cycle***; but the repeated revision that characterizes those later stages start before a piece of software is even born.

1.7 The Interactive Control Loop

This book focuses on the problem of designing, building, and understanding interactive software. The sections above concern basic elements in the design of a computer program. In the next chapter, we will look at the process by which a piece of software actually comes into existence and what happens to it over its life cycle. Although this chapter has focused on design from scratch, most professional software engineers spend most of their time modifying or building on existing code.

Regardless of your particular design problem, you will find it useful to situate your task in the context of these six questions:

- What is the behavior of this program?

If it is a community of entities,

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

And, for each instruction-follower,

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

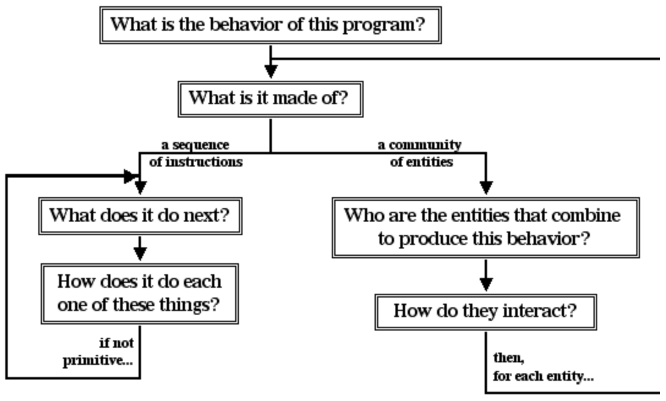
What is the behavior of this program?
What is it made of?
a sequence of instructions
a community of entities
What does it do next?
How does it do each one of these things?
if not primitive...
Who are the entities that combine to produce this behavior?
How do they interact?
then, for each entity...

Figure 1.11. A diagram of the six design questions and their relationships.

At the heart of our approach is the idea of an ***interactive control loop***. This is a simple program that repeatedly receives an input -- a new request, a set of sensor readings, or some other information -- and responds appropriately. An interactive control loop can be just that simple, or it may involve initiating a series of other activities. This kind of program component can be built upon and coupled together to make extremely complex computational systems. Whether simple or complex, the interactive control loop is the crucial bridge that turns an instruction follower into an ongoing participant in an interactive community. In a way, it might be thought of as the "atomic unit" or basic vocabulary element of this kind of computation.

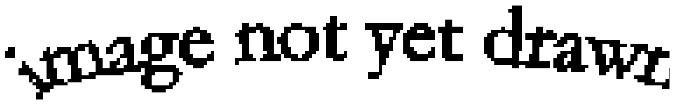

Figure 1.12. Echoer pic.

In concluding this chapter, we will look at one -- extremely simple -- interactive control loop. We will keep this example at hand throughout much of this book and return to it from time to time to explore new variations and themes as we explore interactive programming.

Perhaps the simplest interactive control loop is an ***echo*** program. When run, this program waits for the user to type something. When the user finishes typing, the program simply repeats back what it has been given. That is, it is a loop that gets some input, processes that input (in this case trivially), and then spits out its result. It is roughly the computational equivalent of a mirror.

Although the echo program seems too trivial to be of much use, a minor variant of it runs in almost every program you type to: it is what makes the characters appear on the screen. [[Footnote: In fact, it is what is "turned off" when you type your password, so that no one can see what you type. The setting for *not* showing what you type on the screen is sometimes called "password mode" and sometimes called "no echo".]] Far more importantly, the basic structure of this program underlies essentially every interactive computation. And it demonstrates many of the important properties of an interactive computation:

- It is embedded in an environment (in this case involving a user's typing and a display that the user can see).
- It is interactive (with that user).
- It is concurrent: other things happen at the same time that the program is running. (In this case, the user might be typing the next line even while the echo program is producing its output.)

Let's look at the basic echo program in terms of our six design questions:

- *What is the behavior of this program?*

  It reads in a line from the user and writes that line back to the screen.
- *Who are the entities that combine to produce this behavior?*

  From the outside, it looks like one entity, the echo program. So let's implement it that way. This isn't the only way we might do it, but it's a perfectly reasonable one.

  [[Footnote: It is reasonable to implement the simple echoer as one entity because the job of that entity can be succinctly described and easily implemented. If the job of that entity were to be too complex, we would want to break it down into smaller pieces even though it might look to the outside world like a single entity. This is a design decision that is more art than science, though some of the principles behind it are described in various places throughout this book, but especially in the chapters focusing on object-oriented design (8 - Object Design, 13 - Encapsulation, and 14 - Intelligent Objects and Implicit Dispatch).]]
- *How do they interact?*

Since there's only one, it doesn't interact with anything except the user.

- *What is each one made of? (A community of entities or a single instruction-following control loop?)*

  The one entity -- the echoer -- is an instruction-following control loop.

- *What does it do next?*

  There are two steps:

  1. Read a line from the user.
  2. Write that line to the screen.

- *How does it do each one of these things?*

  Each of these instructions is a built in Java method, i.e., something that Java knows how to do. The specific Java instructions are summarized in the sidebar on the cs101 Console in chapter 3 (Things, Types, and Names).

Now let's look at a slightly more complex design for the echo program. Instead of having one community member -- a single interactive control loop -- we will separate it into a community of two: one that reads the input from the user, and the other that writes the input to the screen. Now, our six design questions will have slightly different answers:

Figure 1.13. Subdivided echoer with reader-in and writer-out.

- *What is the behavior of this program?*

  Still the same as before: it reads in a line from the user and writes that line back to the screen.

- *Who are the entities that combine to produce this behavior?*

  Now, we have two entities: the reader-in and the writer-out.

- *How do they interact?*

  There are several choices we could make here. For now, we'll let the reader-in tell the writer-out each time the reader-in gets a new line. We will explore other answers to this question in part 5 of this book.

- *What is each one made of? (A community of entities or a single instruction-following control loop?)*

  Now, each of our two entities is instruction-following control loop.

- *What does it do next?*

  The reader-in has two steps:

  1. Read a line from the user.
  2. Tell it to the writer-out.

  And the writer-out has two steps:

1. Wait for the reader-in to give you a line.
2. Write that line to the screen.

- *How does it do each one of these things?*

    Again, the instructions are basic things that we can write directly in single lines of Java code.

Of course, this example may seem sort-of forced to illustrate how an echoer can be broken up into pieces and turned into a community. But the division isn't entirely artificial. You see, now that we have separated the echoer into the reader-in and the writer-out, we can do more.

Figure 1.14. Reader-in and writer-out make instant messaging.

For example, we can run the reader-in part of the echoer on your computer and the writer-out part on mine. Now, whatever you type on my computer is echoed not on your screen but on mine. Java provides some pretty simple tools to let the reader-in on your computer talk to the writer-out on my computer[[Footnote: See Streams, which are covered in chapter @@*Streams*@@.]] almost as easily as it can talk to the writer-out on mine. This new, improved echoer is actually doing instant internet messaging!

To really make this idea work, we will of course want to add more features. You should have some control over whose computer you are typing to. I should have some say over whether I want to get instant messages across the internet. And there are plenty of nice graphical user interface features -- windows and buttons and other ways to make the program nicer -- that we probably want to add to our reader-in and to our writer-out. Each of these amounts to changing the way we construct our community, the pieces that interact, and what each one of them is made of, in ways that we will explore through the rest of this book.

The idea of an interactive control loop is the root of this approach to programming. By putting together interactive control loops, you constitute a community of interacting entities. Interactive control loops are what goes inside; communication between them is how they interact. In other words, as they say, all the rest is corollary....

Chapter Summary

@@ needs revision to bring it up to speed with the current version of the text @@

- Computers follow special instructions, called a program, which is written in a special programming language.
- Computation results when a computer has access to these instructions and executes them.
- Each set of instructions must answer:
    - What should the program do next?

    - How should it do it?
- Groups of steps can be combined to make a "higher order" step.
- Steps can involve choices or decisions.
- Steps can be executed over and over again using a loop.
- Most modern programs combine many separate looping instruction-followers into an interacting community.
- Every computation is embedded in an environment and interacts with the other (computational and physical) entities around it.
- The programmer's job is to figure out:
    - What services (behavior) does my program provide?
    - Who are the entities that together provide this behavior?
    - How does each one work?
    - How do they interact?
- @@*use cases*

Exercises

1. Give step by step instructions for how to tie shoelaces.
2. Select your favorite recipe and give step by step instructions for how to cook it.
3. Give detailed directions for how to get from your classroom to where you live. Include indications that will tell whether you've gone too far and how to get back on track.
4. Specify the expected behavior for each of the following interrelated services provided by a bank account:

    a. A deposit.

    b. A withdrawal request.

    c. Checking your balance.

    Does your specification permit overdrafts?

5. You are at a fruit market. Describe the protocol by which you purchase a piece of fruit from the fruit seller.

6. Describe the division of responsibility and coordination of activities among the players on a soccer team.