424f7c283cced86ffd1b25db1e0b8e1b
ebrary

5

High-Level Design

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Design is not just what it looks like and feels like. Design is how it works.

—Steve Jobs

Design is easy. All you do is stare at the screen until drops of blood form on your forehead.

—Marty Neumeier

WHAT YOU WILL LEARN IN THIS CHAPTER:

- The purpose of high-level design
- How a good design lets you get more work done in less time
- Specific things you should include in a high-level design
- Common software architectures you can use to structure an application
- How UML lets you specify system objects and interactions

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

High-level design provides a view of the system at an abstract level. It shows how the major pieces of the finished application will fit together and interact with each other.

A high-level design should also specify assumptions about the environment in which the finished application will run. For example, it should describe the hardware and software you will use to develop the application, and the hardware that will eventually run the program.

The high-level design does not focus on the details of how the pieces of the application will work. Those details can be worked out later during low-level design and implementation.

Before you start learning about specific items that should be part of the high-level design, you should understand the purpose of a high-level design and how it can help you build an application.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

THE BIG PICTURE

You can view software development as a process that chops up the system into smaller and smaller pieces until the pieces are small enough to implement. Using that viewpoint, high-level design is the first step in the chopping up process.

The goal is to divide the system into chunks that are self-contained enough that you could give them to separate teams to implement.

PARALLEL IMPLEMENTATION

Suppose you're building a relatively simple application to record the results of Twister games for a championship. It needs to store the names of the players in each match, the date and time they played, and the order in which they fell over during play.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You might break this application into two large pieces: the database and the user interface. You could then assign those two pieces to different groups of developers to implement in parallel.

(You'll see in the rest of this chapter that there are actually a lot of other pieces you might want to specify even for this simple application.)

There are a lot of variations on this basic theme. On a small project, for example, the project's pieces might be small enough that they can be handled by individual developers instead of teams.

In a large project, the initial pieces might be so big that the teams will want to create their own medium-level designs that break them into smaller chunks before trying to write any code. This can also happen if a piece of the project turns out to be harder than you had expected. In that case, you may want to break it into smaller pieces and assign them to different people.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

ADDING PEOPLE

Breaking an existing task into smaller pieces is one of the few ways you can sometimes add people to a project and speed up development.

Adding new people to the same old tasks usually doesn't help and often actually slows development as the new people get up to speed and get in each other's way. (It can feel like you're in a leaky lifeboat with a single bucket and more people are climbing aboard. You may enjoy the company, but their extra weight will make you sink faster.)

However, if you can break a large task into smaller pieces and assign them to different people, you may speed things up a bit. The new people still need time come up to speed, so this won't always help, but at least people won't trip over each other trying to perform the same tasks.

In some projects, you may want to assign multiple pieces of the project to a single team, particularly if the pieces are closely related. For example, if the pieces pass a lot of data back and forth, it will be helpful if the people building those pieces work closely together. (Multitier architectures, which

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

are described in the "Client/Server" section later in this chapter, can help minimize this sort of interaction.)

Another situation in which this kind of close cooperation is useful is when several pieces of the application all work with the same data structure or with the same database tables. Placing the data structure or tables under the control of a single team may make it easier to keep the related pieces synchronized.

WHAT TO SPECIFY

The stages of a software engineering project often blur together, and that's as true for high-level design as it is for any other part of development. For example, suppose you're building an application to run on the Windows phone platform. In that case, the fact that your hardware platform is Windows phones should probably be in the requirements. (Although you may want to add extra details to the high-level design, such as the models of phones that you will test.)

Exactly what you should specify in the high-level design varies somewhat, but some things are constant for most projects. The following sections describe some of the most common items you might want to specify in the high-level design.

Security

The first thing you see when you start most applications is a login screen. That's the first *obvious* sign of the application's security, but it's actually not the first piece. Before you even log in to the application, you need to log in to the computer.

Your high-level design should sketch out all the application's security needs. Those needs may include the following:

- **Operating system security**—This includes the type of login procedures, password expiration policies, and password standards. (Those annoying rules that say your password must include at least one letter, one number, one special character like # or %, and three Egyptian hieroglyphs.)
- **Application security**—Some applications may rely on the operating system's security and not provide their own. Others may use the operating system's security to make the user reenter the same username and password. Still others may use a separate application username and password. Application security also means providing the right level of access to different users. For example, some users might not be allowed access to every part of the system. (I'll say more about this in the section "User Access" later in the chapter.)
- **Data security**—You need to make sure your customer's credit card information doesn't fall into the hands of Eastern European hackers.
- **Network security**—Even if your application and data are secure, cyber banditos might steal your data from the network.
- **Physical security**—Many software engineers overlook physical security. Your application won't do much good if the laptop it runs on is stolen from an unlocked office.

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

All these forms of security interact with each other, sometimes in non-obvious ways. For example, if you reset passwords too often, users will pick passwords that are easier to remember and possibly easier for hackers to guess. You could add your name to the month number (Rod1 for January, Rod2 for February, and so forth), but those would be easy to guess. If you make the password rules too strict (requiring two characters from each row of the keyboard), users may write their passwords down where they are easy to find.

Physical security also applies to passwords. I've seen large customer service environments in which users often needed manager approval for certain kinds of common operations. In fact, those overrides were so common that the manager didn't have time to handle them and get any other work done. The solution they adopted was to write the manager's username and password on a whiteboard at the front of the room so that everyone could use it to perform their own overrides.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The password was insecure, so any hacker who got into the room could do just about anything with the system. (Fortunately, the room had no windows and was difficult to get into without the right badge and passwords.)

This also meant that any user could impersonate the manager and do just about anything. If that's the case, why bother having user permissions?

If you need to make 50 exceptions per day, then they're not actually exceptions. The solution would have been to not require manager approval for such a common task. Then the manager could have kept her password private and used overrides only for truly important stuff.

Hardware

Back in the old days when programmers worked by candlelight on treadle-powered computers, hardware options were limited. You pretty much wrote computers for large mainframes or desktop computers. You had your pick of a few desktop vendors, and you could pick Windows or Macintosh operating systems, but that was about it.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

These days you have a lot more choices and you need to specify the ones that you'll be using. You can build systems to run on mainframes (yes, they still exist), desktops, laptops, tablets, and phones. Mini-computers act sort of as a mini-mainframe that can serve a handful of users. Personal Digital Assistants (PDAs) are small computers that are basically miniature tablets.

Wearable devices include such gadgets as computers strapped to the wearer's wrist (sort of like a PDA with a wrist strap and possibly extra keys and buttons), wristbands, bracelets, watches, eyeglasses, and headsets.

Additional hardware that you need to specify might include the following:

- Printers
- Network components (cables, modems, gateways, and routers)
- Servers (database servers, web servers, and application servers)
- Specialized instruments (scales, microscopes, programmable signs, and GPS units)
- Audio and video hardware (webcams, headsets, and VOIP)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

With all the available options (and undoubtedly many more on the way), you need to specify the hardware that will run your application. Sometimes, this will be relatively straightforward. For example, your application might run on a laptop or in a web page that could run on any web-enabled hardware. Other times the hardware specification might include multiple devices connected via the Internet, text messages, a custom network, or by some other method.

EXAMPLE Selecting a Hardware Platform

Suppose you're building an application to manage the fleet of dog washing vehicles run by The Pampered Poodle Emergency Dog Washing Service. When a customer calls in to tell you Fifi ran afoul of a skunk, you dispatch an emergency dog-washer to the scene.

In this case, your drivers might access the system over cell phones. A desktop computer back at the office would hold the database and provide a user interface to let you do everything else the business needs such as logging customer calls, dispatching drivers, printing invoices, tracking payments, and ordering doggy shampoo.

For this application, you would specify the kind of phones the drivers will use (such as Windows, iOS, or Android), the model of the computer used to hold the database and business parts of the application, and the type of network connectivity the application will use. (Perhaps the database desktop serves data on the Internet and the phones download data from there.)

Another strategy would be to have the desktop serve information to the drivers as web pages. Then the drivers could use any web-enabled device (smartphone, tablet, Google Glass) to view their assignments.

User Interface

During high-level design, you can sketch out the user interface, at least at a high level. For example, you can indicate the main methods for navigating through the application.

Older-style desktop applications use forms with menus that display other forms. Often the user can display many forms at the same time and switch between them by clicking with the mouse (or touching if the hardware has a touch screen).

In contrast, newer tablet-style applications tend to use a single window (that typically covers the entire tablet, or whatever hardware you're using) and buttons or arrows to navigate. When you click a button, a new window appears and fills the device. Sometimes a Back button lets you move back to the previous window.

Whichever navigational model you pick, you can specify the forms or windows that the application will include. You can then verify that they allow the user to perform the tasks defined in the requirements. In particular, you should walk through the user stories and use cases and make sure you've included all the forms needed to handle them.

In addition to the application's basic navigational style, the high-level user interface design can describe special features such as clickable maps, important tables, or methods for specifying system settings (such as sliders, scrollbars, or text boxes).

This part of the design can also address general appearance issues such as color schemes, company logo placement, and form skins.

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

FOLLOW EXISTING PRACTICES

Most users have a lot of experience with previous applications, and those applications follow certain standardized patterns. For example, desktop applications typically have menus that you access from a form's title bar. The menus drop down below and submenus cascade to the right. That's the way Windows applications have been handling menus for decades and users are familiar with how they work.

If your application sticks to a similar pattern, users will feel comfortable with the application with little extra training. They already know how to use menus, so they won't have any trouble using yours. Instead they can concentrate on learning how to use the more interesting pieces of your system.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Now suppose your application changes this kind of standard interaction. Perhaps you access the menus by clicking a little icon on the right edge of the toolbar and then menus cascade out to the left instead of the right. Or perhaps there are no menus, just panels filled with icons you can click to open new forms. In that case, users will need to learn how to use your new system. That will at least lead to some unnecessary confusion, and it might create a lot of annoyance for the users.

(I use one tool in particular, which I won't name, that for some reason thinks it knows a better way to handle menus, toolbars, and toolboxes. It's frustrating, incredibly annoying, and sometimes leads to major outbreaks of swearing.)

Unless you have a good reason to change the way most applications already work, stick with what the users already know.

You don't need to specify every label and text box for every form during high-level user interface design. You can handle that during low-level design and implementation. (Often the controls you need follow from the database design anyway, so you can sometimes save some work if you do the database design first. Some tools can even use a database design to build the first version of the forms for you.)

Internal Interfaces

When you chop the program into pieces, you should specify how the pieces will interact. Then the teams assigned to the pieces can work separately without needing constant coordination.

It's important that the high-level design specifies these internal interactions clearly and unambiguously so that the teams can work as independently as possible. If two teams that need to interact don't agree on how that interaction should occur, they can waste a huge amount of time. They may waste time squabbling about which approach is better. They will also waste time if one team needs to change the interface and that forces the other team to change its interface, too. The problem increases dramatically if more than two teams need to interact through the same interface.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

It's worth spending some extra time to define these sorts of internal interfaces carefully before developers start writing code. Unfortunately, you may not be able to define the interfaces before writing at least some code. In that case, you may need to insulate two project teams by defining a temporary interface. After the teams have written enough code to know what information they need to exchange, they can define the final interface.

DEFERRED INTERFACES

I worked on one project where two teams needed to pass a bunch of information back and forth. Of course, at the beginning of the project, neither team had written any code to work with the other team, so neither team could call the other. We also weren't sure what data the two teams would need to pass, so we couldn't specify the interface with certainty.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

To get both teams working quickly, the high-level design specified a text file format that the teams could use to load test data. Instead of calling each other's code, the teams could read data from a test data file. They were also free to modify the formats of their files as their needs evolved.

After several months of work, the two teams had written code to process the data and their needs were better defined. At that point, they agreed on a format for passing data and switched from loading data from data files to actually calling each other's code.

It would have been more efficient to have defined the perfect interface at the beginning during high-level design, but that wasn't an option. Using text files to act as temporary interfaces allowed both teams to work independently.

(The multitier design described in the "Architecture" section later in this chapter does something similar.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

External Interfaces

Many applications must interact with external systems. For example, suppose you're building a program that assigns crews for a large chartered fishing company. The application needs to assign a captain, first mate, and cook for each trip. Your program needs to interact with the existing employee database to get information about crew members. (You don't want to assign a boat three cooks and no captain.) You might also need to interact with a sales program that lets salespeople book fishing trips.

In a way, external interfaces are often easier to specify than internal ones because you usually don't have control over both ends of the interface. If your application needs to interact with an existing system, then that system already has interface requirements that you must meet.

Conversely, if you want future systems to interface with yours, you can probably specify whatever interface makes sense to you. Systems developed later need to meet your requirements. (Try to make your interface simple and flexible so that you don't get flooded with change requests.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Architecture

An application's architecture describes how its pieces fit together at a high level. Developers use a lot of "standard" types of architectures. Many of these address particular characteristics of the problem being solved.

For example, rule-base systems are often used to handle complex situations in which solving a particular problem can be reduced to following a set of rules. Some troubleshooting systems use this approach. You call in because your computer can't connect to the Internet, and a customer rep from some distant time zone asks you a sequence of questions to try to diagnose the problem. The rep reads a question off a computer screen, you answer, and the rep clicks the corresponding button to get to the next question. Rules inside the rep's diagnostic system decide which question to give you next.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Other architectures attempt to simplify development by reducing the interactions among the pieces of the system. For example, a component-based architecture tries to make each piece of the system as separate as possible so that different teams of developers can work on them separately.

The following sections describe some of the most common architectures.

Monolithic

In a *monolithic architecture*, a single program does everything. It displays the user interface, accesses data, processes customer orders, prints invoices, launches missiles, and does whatever else the application needs to do.

This architecture has some significant drawbacks. In particular, the pieces of the system are tied closely together, so it doesn't give you a lot of flexibility. For example, suppose the application stores customer address data and you later need to change the address format. (Perhaps you add a field to hold suite numbers.) Then you also need to change every piece of code that uses the address. This may not be too hard, but it means the programmers working on related pieces of code must stop what they're doing and deal with the change before they can get back to their current tasks. (The multitier architectures described in the next section handle this better, allowing the different teams of developers to work more independently.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

A monolithic architecture also requires that you understand how all the pieces of the system fit together from the beginning of the project. If you get any of the details wrong, the tight coupling between the pieces of the system makes fixing them later difficult.

Monolithic architectures do have some advantages. Because everything is built into a single program, there's no need for complicated communication across networks. That means you don't need to write and debug communication routines; you don't need to worry about the network going down; and you don't need to worry about network security. (Well, you still need to worry about some hacker sneaking in through your network and attacking your machines, but at least you don't need to encrypt messages sent between different parts of the application.)

Monolithic architectures are also useful for small applications where a single programmer or team is working on the code.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Client/Server

A *client/server architecture* separates pieces of the system that need to use a particular function (clients) from parts of the system that provide those functions (servers). That decouples the client and server pieces of the system so that developers can work on them separately.

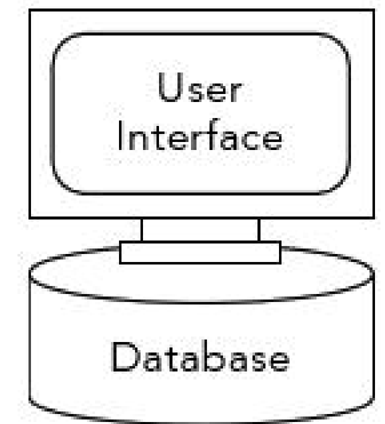


FIGURE 5-1: An application can directly hold its own data.

For example, many applications rely on a database to hold information about customers, products, orders, and employees. The application needs to display that information in some sort of user interface. One way to do that would be to integrate the database directly into the application. Figure 5-1 shows this situation schematically.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

One problem with this design is that multiple users cannot use the same data. You can fix that problem by moving to a *two-tier architecture* where a client (the user interface) is separated from the server (the database). Figure 5-2 shows this design. The clients and server communicate through some network such as a local area network (LAN), wide area network (WAN), or the Internet.

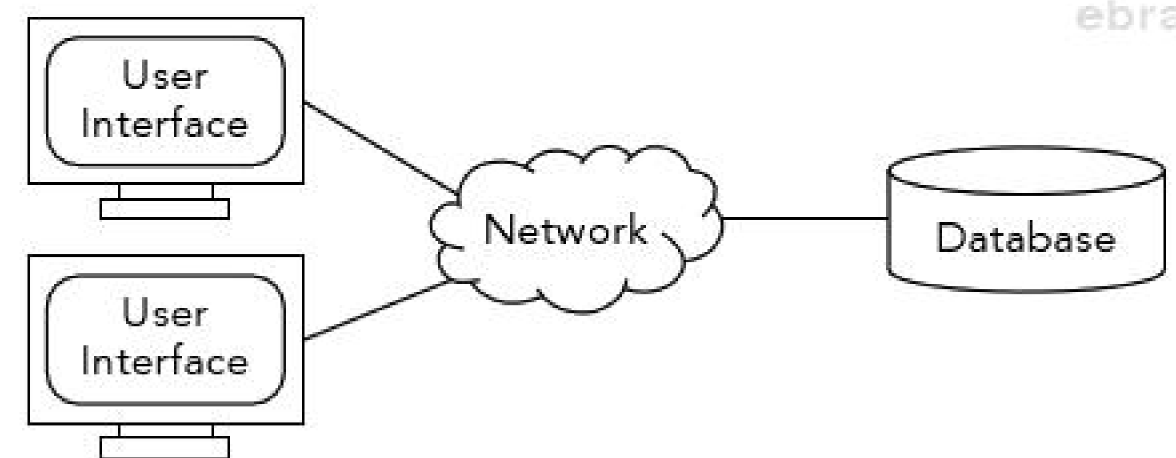


FIGURE 5-2: In a two-tier architecture, the client is separate from the server.

In this example, the client is the user interface (two instances of the same program) and the server is a database, but that need not be the case. For example, the client could be a program that makes automatic stock purchases, and the server could be a program that scours the Internet for information about companies and their stocks.

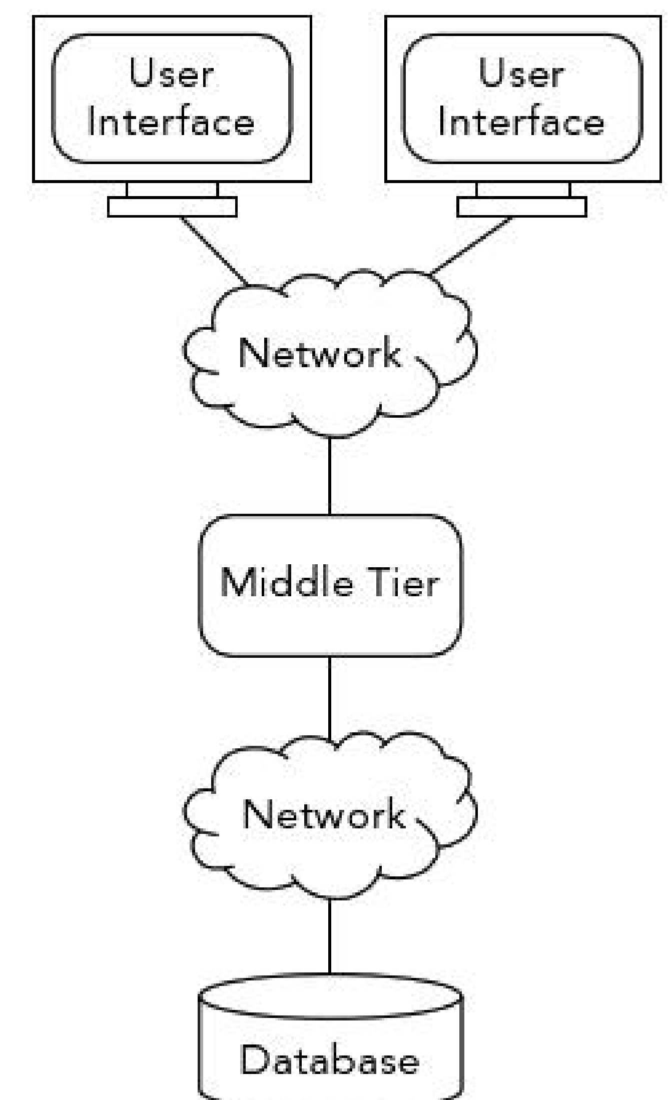


FIGURE 5-3: A three-tier architecture separates clients and servers with a middle tier.

The two-tier architecture makes it easier to support multiple clients with the same server, but it ties clients and servers relatively closely together. The clients must know what format the server uses, and if you change the way the server presents its data, you need to change the client to match. That may not always be a big problem, but it can mean a lot of extra work, particularly in the beginning of a project when the client's and server's needs aren't completely known.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You can help to increase the separation between the clients and server if you introduce another layer between the two to create the *three-tier architecture*, as shown in Figure 5-3.

In Figure 5-3, the middle tier is separated from the clients and the server by networks. The database runs on one computer, the middle tier runs on a second computer, and the instances of the client run on still other computers. This isn't the only way in which the pieces of the system can communicate. For example, in many applications the middle tier runs on the same computer as the database.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In a three-tier architecture, the middle tier provides insulation between the clients and server. In this example, it provides an interface that can map data between the format provided by the server and the format needed by the client. If you need to change the way the server stores data, you need to update only the middle tier so that it translates the new format into the version expected by the client.

Conversely, if the client's data needs change, you can modify the middle tier to insert fake data until you have a chance to update the server to provide the actual data.

The separation provided by the middle tier lets different teams work on the client and server without interfering with each other too much.

In addition to providing separation, a middle tier can perform other actions that make the data easier to use by the client and server. For example, suppose the client needs to display some sort of aggregate data. Perhaps Martha's Musical Mechanisms needs to display the total number of carillons sold by each employee for each of the last 12 quarters. In that case, the server could store the raw sales data, and the middle tier could aggregate the data before sending it to the client.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TIER TERMINOLOGY

Sometimes, the client tier is called the *presentation tier* (because it presents information to the user); the middle tier is called the *logic tier* (because it contains business logic such as aggregating data for the presentation tier); and the client tier is called the *data tier* (particularly if all it does is provide data).

You can define other *multitier architectures* (or *N-tier architectures*) that use more than three tiers if that would be helpful. For example, a data tier might store the data, a second tier might calculate aggregates and perform other calculations on the data, a third tier might use artificial intelligence techniques to make recommendations based on the second tier's data, and a fourth tier would be a presentation tier that lets users see the results.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

BEST PRACTICE

Multitier architectures are a best practice, largely because of the separation they provide between the client and server layers. Most applications don't use more than three tiers.

Component-Based

In *component-based software engineering* (*CBSE*), you regard the system as a collection of loosely coupled components that provide services for each other. For example, suppose you're writing a system to schedule employee work shifts. The user interface could dig through the database to see what hours are available and what hours an employee can work, but that would tie the user interface closely to the database's structure.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

An alternative would be to have the user interface ask components for that information, as shown in Figure 5-4. (UML provides a more complex diagram for services that is described in the section "UML" later in this chapter.)

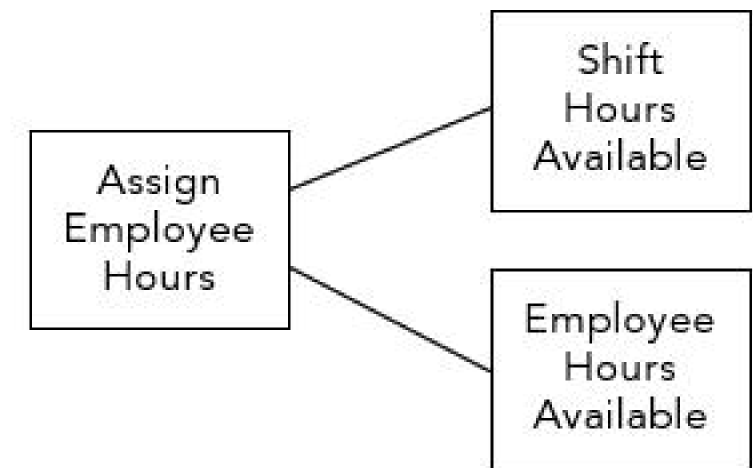


FIGURE 5-4: In a component-based architecture, components help decouple pieces of code.

The Assign Employee Hours user interface component would use the Shift Hours Available component to find out what hours were not yet assigned. It would use the Employee Hours Available component to find out what hours an employee has available. After assigning new hours to the employee, it would update the other two components so that they know about the new assignment.

A component-based architecture decouples the pieces of code much as a multitier architecture does, but the pieces are all contained within the same executable program, so they communicate directly instead of across a network.

Service-Oriented

A *service-oriented architecture* (*SOA*) is similar to a component-based architecture except the pieces are implemented as services. A *service* is a self-contained program that runs on its own and provides some kind of service for its clients.

Sometimes, services are implemented as *web services*. Those are simply programs that satisfy certain standards, so they are easy to invoke over the Internet.

DEFINING SOA

Some big software vendors such as IBM and Oracle also define *Service Component Architecture* (*SCA*). This is basically a set of specifications for SOA defined by those companies.

Data-Centric

Data-centric or *database-centric architectures* come in a variety of flavors that all use data in some central way. The following list summarizes some typical data-centric designs:

- Storing data in a relational database system. This is so common that it's easy to think of as a simple technique for use in other architectures rather than an architecture of its own.
- Using tables instead of hard-wired code to control the application. Some artificial intelligence applications such as rule-based systems use this approach.
- Using stored procedures inside the database to perform calculations and implement business logic. This can be a lot like putting a middle tier inside the database.

Event-Driven

In an *event-driven architecture* (*EDA*), various parts of the system respond to events as they occur. For example, as a customer order for robot parts moves through its life cycle, different pieces of the system might respond at different times. When the order is created, a fulfillment module might

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

notice and print a list of the desired parts and an address label. When the order has been shipped, an invoicing module might notice and print an invoice. When the customer hasn't paid the invoice for 30 days, an enforcement module might notice and send RoboCop to investigate.

Rule-Based

A *rule-based architecture* uses a collection of rules to decide what to do next. These systems are sometimes called *expert systems* or *knowledge-based systems.*

The troubleshooting system described earlier in this chapter uses a rule-based approach.

Rule-based systems work well if you can identify the rules necessary to get the job done. Sometimes, you can build good rules even for complicated systems; although that can be a lot of work.

Rule-based systems don't work well if the problem is poorly defined so you can't figure out what rules to use. They also have trouble handling unexpected situations.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

ROTTEN RULES

For several years I had a fairly odd network connection leading directly to my phone company's central office. One day it didn't work, so I called tech support, and the service rep started working through his troubleshooting rules. Unfortunately, the phone company hadn't offered my type of service for several years, so the rules didn't cover it.

Eventually, the rep reached a rule that asked me to unplug my modem and reconnect it. I explained that the modem was in the central office and that unplugging anything on my end would also disconnect my phone. The rules didn't give him any other options, so he insisted. I unplugged my cable and predictably the phone call dropped.

I called back, got a different rep who was a little better at thinking outside of the rules, and we discovered (as I had suspected) that the problem was at the central office.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Rule-based systems are great for handling common simple scenarios, but when they encounter anything unexpected they're quite useless. For that reason, you should always give the user a way to handle special situations manually.

Distributed

In a *distributed architecture*, different parts of the application run on different processors and may run at the same time. The processors could be on different computers scattered across the network, or they could be different cores on a single computer. (Most modern computers have multiple cores that can execute code at the same time.)

Service-oriented and multitier architectures are often distributed, with different parts of the system running on different computers. Component-oriented architectures may also be distributed, with different components running on different cores on the same computer.

In general, distributed applications can be extremely confusing and hard to debug. For example, suppose you're writing an application that sells office supplies such as staples, paper

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

clips, and demotivational posters. You sell to companies that might have several authorized purchasers.

Now suppose your application uses the following steps to add the cost of a new purchase to a customer's outstanding balance:

1. Get customer balance from database.
2. Add new amount to balance.
3. Save new balance in database.

This seems straightforward until you think about what happens if two people make purchases at almost the same time with a distributed application. Suppose a customer has an outstanding balance of $100. One purchaser buys $50 worth of sticky notes while another purchaser is buying a $10 trash can labeled "suggestions." Now suppose the application executes the two purchasers' steps in the order shown in Table 5-1.

TABLE 5-1: Office Supply Purchasing Sequence

PURCHASER 1	PURCHASER 2
Get balance. ($100)	
	Get balance. ($100)
Add to balance. ($150)	
	Add to balance. ($110)
Save new balance. ($150)	
	Save new balance. ($110)

In Table 5-1, time increases downward so Purchaser 1 gets the account balance first and then Purchaser 2 gets the account balance.

Next Purchaser 1 adds $50 to his balance to get $150, and then Purchaser 2 adds $10 to his balance to get $110.

Purchaser 1 then saves his new balance of $150 into the database. Finally Purchaser 2 saves his balance of $110 into the database, writing over the $150-balance that Purchaser 1 just saved. In the end, instead of holding a balance of $160 ($100 + $50 + $10), the database holds a balance of $110.

In distributed computing, this is called a *race condition*. The two processes are racing to see which one saves its balance first. Whichever one saves its balance second "wins." (Although you lose.)

A distributed architecture can improve performance as long as you don't run afoul of race conditions and other potential problems.

Mix and Match

An application doesn't need to stick with a single architecture. Different pieces of the application might use different design approaches. For example, you might create a distributed service-oriented

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

application. Some of the larger services might use a component-based approach to break their code into decoupled pieces. Other services might use a multitier approach to separate their features from the data storage layer. (Combining different architectures can also sound impressive at cocktail parties. "Yes, we decided to go with an event-driven multitier approach using rule-based distributed components.")

CLASSYDRAW ARCHITECTURE

Suppose you want to pick an architecture for the ClassyDraw application described in Chapter 4. (Recall that this is a drawing program somewhat similar to MS Paint except it lets you select and manipulate drawing objects.) One way to do that is to think about each of the standard architectures and decide whether it would make sense to use while building the program.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

1. **Monolithic**—This is basically the default if none of the more elaborate architectures apply. We'll come back to this one later.
2. **Client/server, multitier**—ClassyDraw stores drawings in files, not a database, so client/server and multitier architectures aren't needed. (You could store drawings in a database if you wanted to, perhaps for an architectural firm or some other use where there would be some benefit. For a simple drawing application, it would be overkill.)
3. **Component-based**—You could think of different pieces of the application as components providing services to each other. For example, you could think of a "rectangle component" that draws a rectangle. For this simple application, it's probably just as easy to think of a `Rectangle` class that draws a rectangle, so I'm not going to think of this as a component-based approach.
4. **Service-oriented**—This is even less applicable than the component-based approach. Spreading the application across multiple computers connected via web services (or some other kind of service) wouldn't help a simple drawing application.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

5. **Data-centric**—The user defines the drawings, so there's no data around which to organize the program. (Although a more specialized program, perhaps a drafting program for an architectural firm or an aerospace design program, might interact with data in a meaningful way.)
6. **Event-driven**—The user interface will be event-driven. For example, the user selects a tool and then clicks and drags to create a new shape.
7. **Rule-based**—There are no rules that the user must follow to make a drawing, so this program isn't rule-based.
8. **Distributed**—This program doesn't perform extensive calculations, so distributing pieces across multiple CPUs or cores probably wouldn't help.

Because none of the more exotic architectures applied (such as multitier or service-oriented), this application can have a simple monolithic architecture with an event-driven user interface.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Reports

Almost any nontrivial software project can use some kinds of reports. Business applications might include reports that deal with customers (who's buying, who has unpaid bills, where customers live), products (inventory, pricing, what's selling well), and users (which employees are selling a lot, employee work schedules).

Even relatively simple applications can sometimes benefit from reports. For example, suppose you're writing a simple shareware game that users will download from the Internet and install on their phones. The users won't want reports (except perhaps a list of their high scores), but you may want to add some reporting. You could make the game upload information such as where the users are, when they use the game, how often they play, what parts of the game take a long time, and so forth. You can then use that data to generate reports to help you refine the game and improve your marketing.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

AD HOC REPORTING

A large application might have dozens or even hundreds of reports. Often customers can give you lists of existing reports that they use now and that they want in the new system. They may also think of some new reports that take advantage of the new system's features.

However, as development progresses, customers inevitably think of more reports as they learn more about the system. They'll probably even think of extra reports after you've completely finished development.

Adding dozens of new reports throughout the development cycle can be a burden to the developers. One way to reduce report proliferation is to forbid it. Just don't allow the customers to request new reports. Or you could allow new reports but require that they go through some sort of approval process so you don't get too many requests.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Another approach is to allow the users to create their own reports. If the application uses a SQL database, it's not too hard to buy or build a reporting tool that lets users type in queries and see the results. I've worked on projects where the customers used this capability to design dozens of new reports without creating extra work for the developers.

If you use this technique, however, you may need to restrict access to it so the users don't see confidential data. For example, a typical order entry clerk probably shouldn't be able to generate a list of employee salaries.

Some SQL statements can also damage the database. For example, the SQL `DROP TABLE` statement can remove a table from the database, destroying all its data. Make sure the ad hoc reporting tool is only usable by trusted users or that it won't allow those kinds of dangerous commands.

As is the case with high-level user interface design, you don't need to specify every detail for every report here. Try to decide which reports you'll need and leave the details for low-level design and implementation.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Other Outputs

In addition to normal reports, you should consider other kinds of outputs that the application might create. The application could generate printouts (of reports and other things), web pages, data files, image files, audio (to speakers or to audio files), video, output to special devices (such as electronic signs), e-mail, or text messages (which is as easy as sending an e-mail to the right address). It could even send messages to pagers, if you can find any that aren't in museums yet.

> **TIP** *Text (or pager) messages are a good way to tell operators that something is going wrong with the application. For example, if an order processing application is stuck and jobs are piling up in a queue, the application can send a message to a manager, who can then try to figure out what's wrong.*

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Database

Database design is an important part of most applications. The first part of database design is to decide what kind of database the program will need. You need to specify whether the application will store data in text files, XML files, a full-fledged relational database, or something more exotic such as a temporal database or object store. Even a program that doesn't use any database still needs to store data, perhaps inside the program within arrays, lists, or some other data structure.

If you decide to use an external database (in other words, more than data that's built into the code), you should specify the database product that you will use. Many applications store their data in relational databases such as Access, SQL Server, Oracle, or MySQL. (There are dozens if not hundreds of others.)

If you use a relational database, you can sketch out the tables it contains and their relationships during high-level design. Later you can provide more details such as the specific fields in each table and the fields that make up the keys linking the tables.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

DEFINING CLASSES

Often the tables in the database correspond to classes that you need to build in the code. At this point, it makes sense to write down any important classes you define. Those might include fairly obvious classes such as `Employee`, `Customer`, `Order`, `WorkAssignment`, and `Report`.

You'll have a chance to refine those classes and add others during low-level design and implementation. For example, you might create subclasses that add refinement to the basic high-level classes. You could create subclasses of the `Customer` class such as `PreferredCustomer`, `CorporateCustomer`, and `ImpulseBuyer`.

Use good database design practices to ensure that the database is properly normalized. Database design and normalization is too big a topic to cover in this book. (For an introduction to database design, see my book *Beginning Database Design Solutions*, Wiley, 2008.) Although

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

I don't have room to cover those topics in depth, I'll say more about normalization in the next chapter.

Meanwhile there are three common database-specific issues that you should address during high-level design: audit trails, user access, and database maintenance.

Audit Trails

An *audit trail* keeps track of each user who modifies (and in some applications views) a specific record. Later, management can use the audit trails to see which employee gave a customer a 120-percent discount. Auditing can be as simple as creating a history table that records a user's name, a link to the record that was modified, and the date when the change occurred. Some database products can even create audit trails for you.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

A fancier version might store copies of the original data in each table when its data is modified. For example, suppose a user changes a customer's billing data to show the customer paid in full. Instead of updating the customer's record, the program would mark the existing (unpaid) record as outdated. It would then copy the old record, update it to show the customer's new balance, and add the date of the change and the user's name. Some applications also provide space for the users to add a note explaining why they gave the customer a $12,000-credit on the purchase of a box of cereal.

Later, you can compare the customer's records over time to build an audit trail that re-creates the exact sequence of changes made for that customer. (Of course, that means you need to add a way for the application to display the audit trail, and that means more work.)

> **NOTE** *Some businesses have rules or government regulations that require them to delete old data including audit trails.*

Many applications don't need auditing. If you write an online multiplayer rock-paper-scissors game, you probably don't need an extensive record of who picked paper in a match two months ago. You also may not need to add auditing to programs written for internal company use, and other programs that don't involve money, confidential records, or other data that might be tempting to misuse. In cases like those, you can simplify the application by skipping audit trails.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

User Access

Many applications also need to provide different levels of access to different kinds of data. For example, a fulfillment clerk (who throws porcelain dishes into a crate for shipping) probably doesn't need to see the customer's billing information, and only managers need to see the other employees' salary information.

One way to handle user access is to build a table listing the users and the privileges they should be given. The program can then disable or remove the buttons and menu items that a particular user shouldn't be allowed to use.

Many databases can also restrict access to tables or even specific columns in tables. For example, you might be able to allow all users to view the `Name`, `Office`, and `PhoneNumber` fields in the `Employees` table without letting them see the `Salary` field.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Database Maintenance

A database is like a hall closet: Over time it gets disorganized and full of random junk like string, chipped vases, and unmatched socks. Every now and then, you need to reorganize so that you can find things efficiently.

If you use audit trails and the records require a lot of changes, the database will start to fill up with old versions of records that have been modified. Even if you don't use audit trails, over time the database can become cluttered with outdated records. You probably don't need to keep the records of a customer's gum purchase three years ago.

In that case, you may want to move some of the older data to long-term storage to keep the main database lean and responsive. Depending on the application, you may also need to design a way to retrieve the old data if you decide you want it back later.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You can move the older data into a *data warehouse*, a secondary database that holds older data for analysis. In some applications, you may want to analyze the data and store modified or aggregated forms in the warehouse instead of keeping every outdated record.

You may even want to discard the old data if you're sure you'll never need it again.

Removing old data from a database can help keep it responsive, but a lot of changes to the data can make the database's indexes inefficient and that can hurt performance. For that reason, you may need to periodically re-index key tables or run database tuning software to restore peak performance. In large, high-reliability applications, you might need to perform these sorts of tasks during off-peak hours such as between midnight and 2 a.m.

Finally, you should design a database backup and recovery scheme. In a low-priority application, that might involve copying a data file to a DVD every now and then. More typically, it means copying the database every night and saving the copy for a few days or a week. For high-reliability systems, it may mean buying a special-purpose database that automatically shadows every change made to any database record on multiple computers. (One telephone company project I worked on even required the computers to be in different locations so that they wouldn't all fail if a computer room was flooded or wiped out by a tornado.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

These kinds of database maintenance activities don't necessarily require programming, but they're all part of the price you pay for using big databases, so you need to plan for them.

Configuration Data

I mentioned earlier that you can save yourself a lot of time if you let users define their own ad hoc queries. Similarly, you can reduce your workload if you provide configuration screens so that users can fine-tune the application without making you write new code. Store parameters to algorithms, key amounts, and important durations in the database or in configuration files.

For example, suppose your application generates late payment notices if a customer has owed at least $50 for more than 30 days. If you make the values $50 and 30 days part of the configuration, you won't need to change the code when the company decides to allow a 5-day grace period and start pestering customers only after 35 days.

Make sure that only the right users can modify the parameters. In many applications, only managers should change these values.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Data Flows and States

Many applications use data that flows among different processes. For example, a customer order might start in an Order Creation process, move to Order Assembly (where items are gathered for shipping), and then go to Shipping (for actual shipment). Data may flow from Shipping to a final Billing process that sends an invoice to the customer via e-mail. Figure 5-5 shows one way you might diagram this data flow.

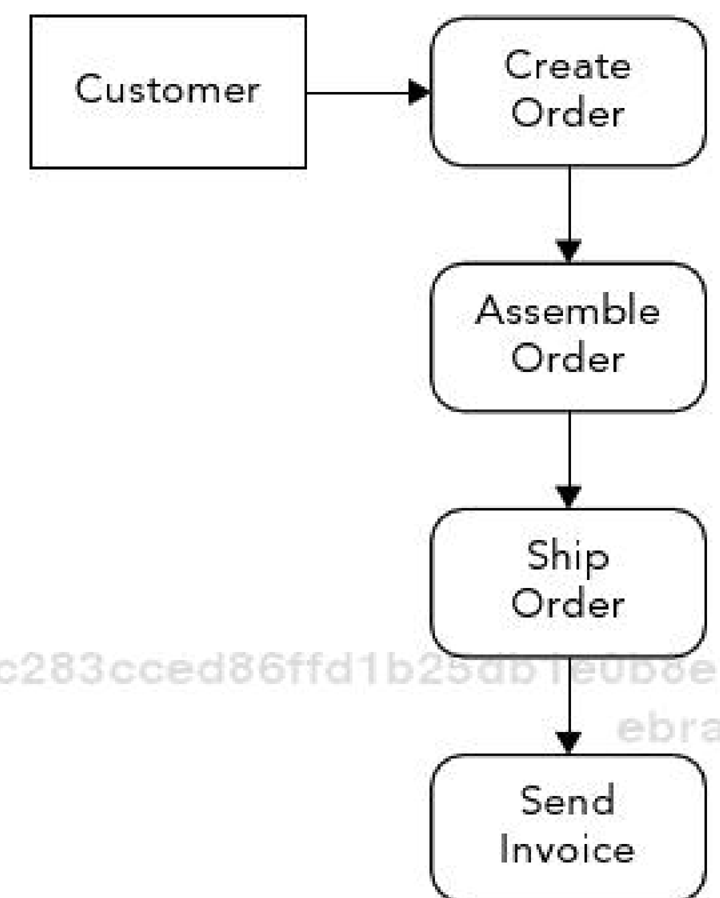


FIGURE 5-5: A data flow diagram shows how data such as a customer order flows through various processes.

You can also think of a piece of data such as a customer order as moving through a sequence of states. The states often correspond to the processes in the related data flow. For this example, a customer order might move through the states Created, Assembled, Shipped, and Billed.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Not all data flows and state transitions are as simple as this one. Sometimes events can make the data take different paths through the system. Figure 5-6 shows a state transition diagram for a customer order. The rounded rectangles represent states. Text next to the arrows indicates events that drive transitions. For example, if the customer hasn't paid an invoice 30 days after the order enters the Billed state, the system sends a second invoice to the customer and moves the order to the late state.

These kinds of diagrams help describe the system and the way processes interact with the data.

Training

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Although it may not be time to start writing training materials, it's never too early to think about them. The details of the system will probably change a lot between high-level design and final installation, but you can at least think about how you want training to work. You can decide whether you want users to attend courses taught by instructors, read printed manuals, watch instructional videos, or browse documentation online.

Trainers may create content that discusses the application's high-level purpose, but you have to fill in most of the details later as the project develops.

UML

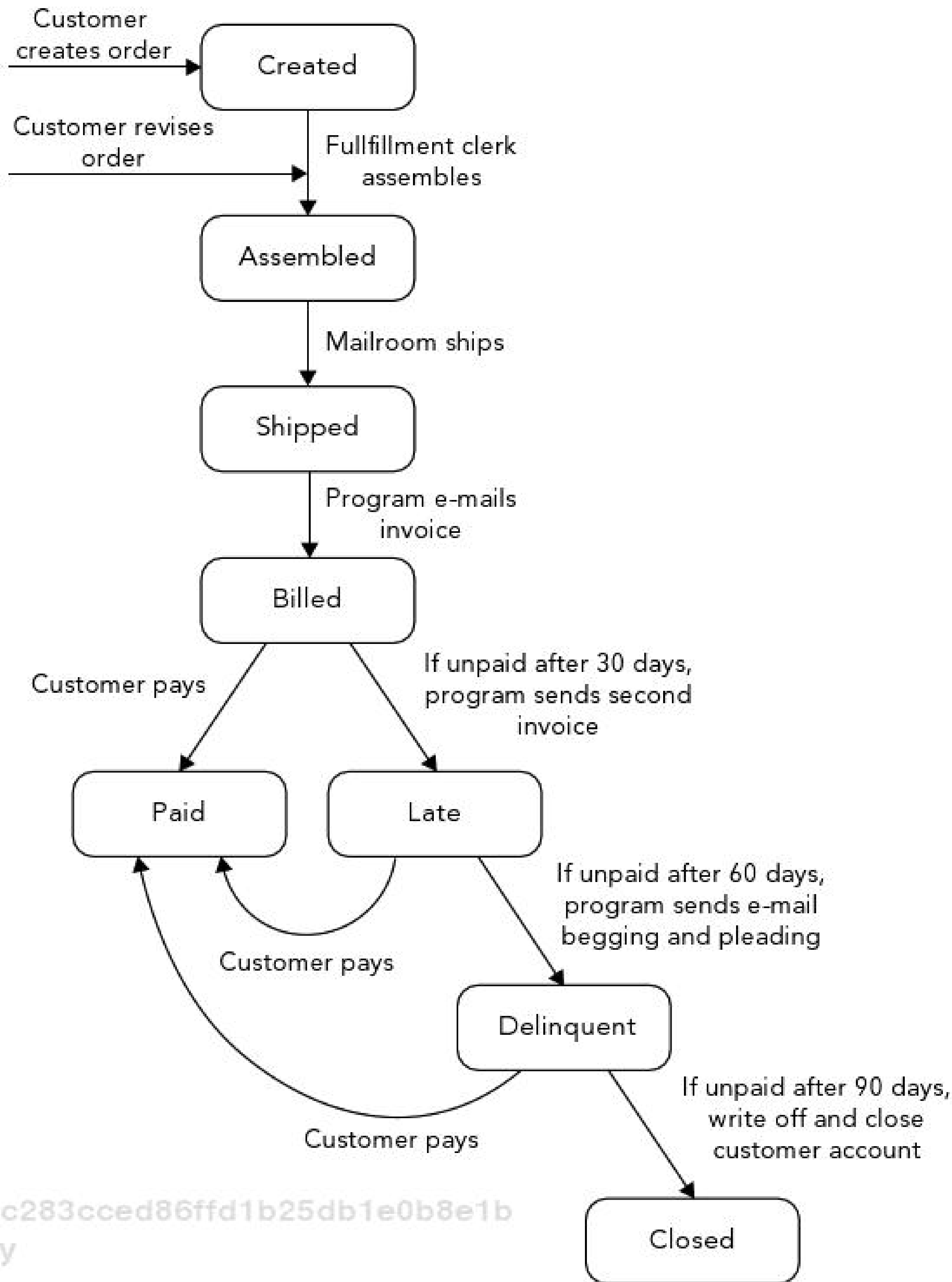
As mentioned in Chapter 4, "Requirement Gathering," the Unified Modeling Language (UML) isn't actually a single unified language. Instead it defines several kinds of diagrams that you can use to represent different pieces of the system.

The Object Management Group (OMG, yes, as in "OMG how did they get such an awesome acronym before anyone else got it?") is an international not-for-profit organization that defines modeling standards including UML. (You can learn more about OMG and UML at `www.uml.org`.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary



424f7c283cced86ffd1b25db1e0b8e1b
ebrary

FIGURE 5-6: A data flow diagram shows how data such as a customer order flows through various processes.

UML 2.0 defines 13 diagram types divided into three categories (and one subcategory) as shown in the following list:

- Diagram
  - Structure Diagram
    - Class Diagram
    - Composite Structure Diagram
    - Component Diagram
    - Deployment Diagram

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

        - Object Diagram
        - Package Diagram
        - Profile Diagram
    - Behavior Diagram
        - Activity Diagram
        - Use Case Diagram
        - State Machine Diagram
        - Interaction Diagram
            - Sequence Diagram
            - Communication Diagram
            - Interaction Overview Diagram
            - Timing Diagram

Many of these are rather complicated so I won't describe them all in excruciating detail here. Instead the following sections give overviews of the types of diagrams in each category and provide a bit more detail about some of the most commonly used diagrams.

Structure Diagrams

A *structure diagram* describes things that will be in the system you are designing. For example, the class diagram (one type of structure diagram) shows relationships among the classes that will represent objects in the system such as inventory items, vehicles, expense reports, and coffee requisition forms.

OBJECTS AND CLASSES

I'll say a bit more about classes and class diagrams shortly, but briefly a *class* defines a type (or class) of items, and an *object* is an instance of the class. Often classes and objects correspond closely to real-world objects.

For example, a program might define a `Student` class to represent students. The class would define properties that all students share such as `Name`, `Grade`, and `HomeRoom`.

A specific instance of the `Student` class would be an object that represents a particular student, such as Rufus T. Firefly. For that object, the `Name` property would be set to "Rufus T. Firefly," `Grade` might be 12, and `HomeRoom` might be "11-B."

The following list summarizes UML's structure diagrams:

- Class Diagram—Describes the classes that make up the system, their properties and methods, and their relationships.
- Object Diagram—Focuses on a particular set of objects and their relationships at a specific time.

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

- Component Diagram—Shows how components are combined to form larger parts of the system.
- Composite Structure Diagram—Shows a class's internal structure and the collaborations that the class allows.
- Package Diagram—Describes relationships among the packages that make up a system. For example, if one package in the system uses features provided by another package, then the diagram would show the first "importing" the second.
- Deployment Diagram—Describes the deployment of ***artifacts*** (files, scripts, executables, and the like) on ***nodes*** (hardware devices or execution environments that can execute artifacts).

The most basic of the structure diagrams is the class diagram. In a class diagram, a class is represented by a rectangle. The class's name goes at the top, is centered, and is in bold. Two sections below the name give the class's properties and methods. (A ***method*** is a routine that makes an object do something. For example, the `Student` class might have a `DoAssignment` method that makes the `Student` object work through a specific class assignment.) Figure 5-7 shows a simple diagram for the `Student` class.

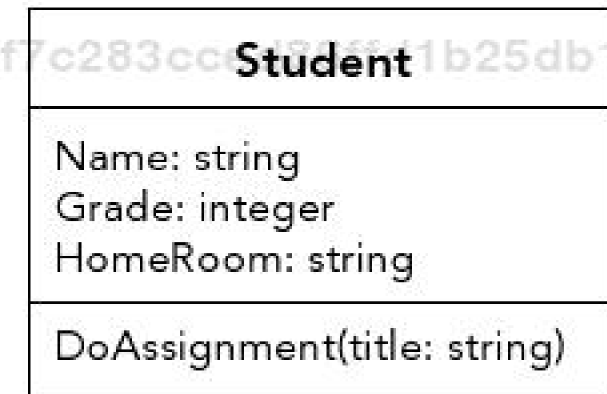


FIGURE 5-7: A class diagram describes the properties and methods of classes.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Some people add annotations to class representations to give you more detail. Most class diagrams include the data types of properties and parameters passed into methods, as shown in Figure 5-7. You can also add the symbols shown in Table 5-2 to the left of a class member to show its visibility within the project.

TABLE 5-2: Class Diagram Visibility Symbols

SYMBOL	MEANING	EXPLANATION
+	Public	The member is visible to all code in the application.
–	Private	The member is visible only to code inside the class.
#	Protected	The member is visible only to code inside the class and any derived classes.
~	Package	The member is visible only to code inside the same package.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Class diagrams also often show relationships among classes. Lines connect classes that are related to each other. A variety of line styles, symbols, arrowheads, and annotations give more information about the kinds of relationships.

The simplest way to use relationships is to draw an arrow indicating the direction of the relationship and label the arrow with the relationship's name. For example, in a school registration application, you might draw an arrow from the `Student` class to the `Course` class to indicate that a `Student` is associated with the `Courses` that student is taking. You could label that arrow "is taking."

At the line's endpoints, you can add symbols to indicate how many objects are involved in the relationship. Table 5-3 shows symbols you can add to the ends of a relationship.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TABLE 5-3: Class Diagram Multiplicity Indicators

SYMBOLS	MEANING
1	Exactly 1
0..1	0 or 1
0..*	Any number (0 or more)
*	Any number (0 or more)
1..*	1 or more

The class diagram in Figure 5-8 shows the "is taking" relationship between the `Student` and `Course` classes. In that relationship, 1 `Student` object corresponds to 1 or more `Course` objects.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

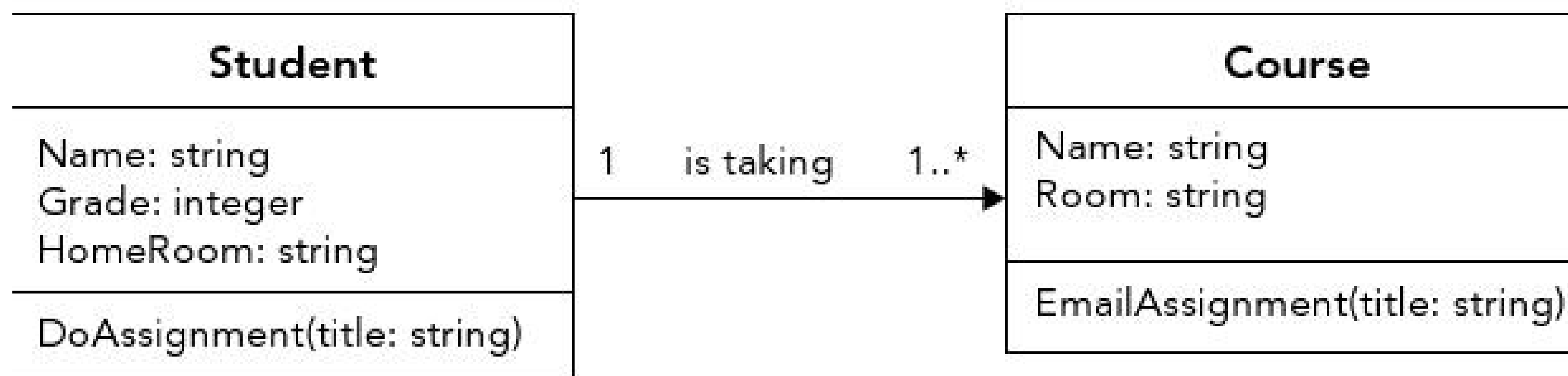


FIGURE 5-8: The relationship in this class diagram indicates that 1 Student takes 1 or more Courses.

Another important type of class diagram relationship is inheritance. In object-oriented programming, one class can inherit the properties and methods of another. For example, an honors student is a type of student. To model that in an object-oriented program, you could define an `HonorsStudent` class that inherits from the `Student` class. The `HonorsStudent` class automatically gets any properties and methods defined by the `Student` class (`Name`, `Grade`, `HomeRoom`, and `DoAssignment`). You can also add new properties and methods if you like. Perhaps you want to add a `GPA` property to the `HonorsStudent` class.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In a class diagram, you indicate inheritance by using a hollow arrowhead pointing from the child class to the parent class. Figure 5-9 shows that the `HonorsStudent` class inherits from the `Student` class.

Class diagrams for complicated applications can become cluttered and hard to read if you put everything in a single huge diagram. To reduce clutter, developers often draw multiple class diagrams showing parts of the system. In particular, they often make separate diagrams to show inheritance and other relationships.

For information about more elaborate types of class diagrams, search the Internet in general or the OMG website `www.omg.org` in particular.

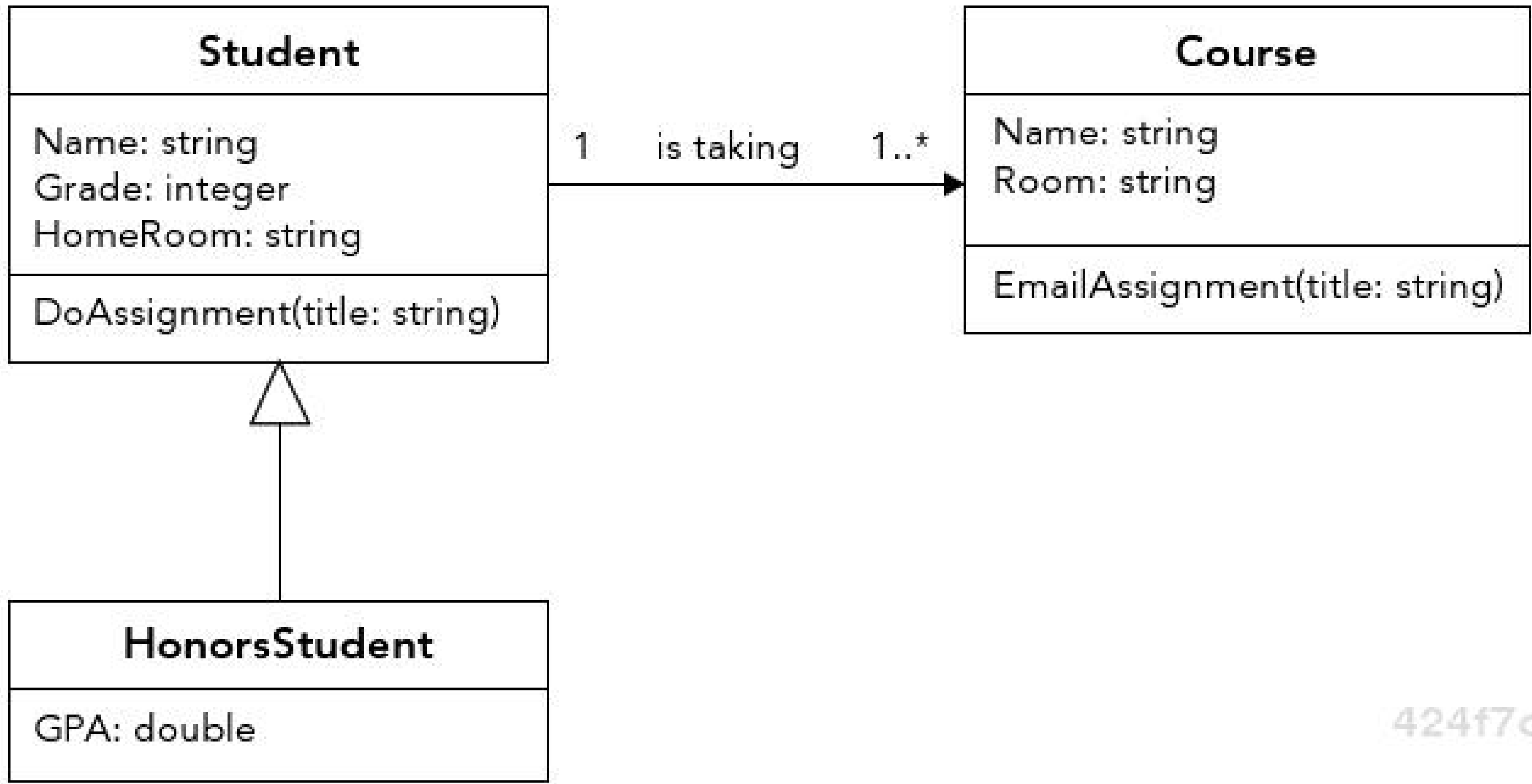
Behavior Diagrams

UML defines three kinds of basic *behavior diagrams*: activity diagrams, use case diagrams, and state machine diagrams. The following sections provide brief descriptions of these kinds of diagrams and give a few simple examples.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary



424f7c283cced86ffd1b25db1e0b8e1b
ebrary

FIGURE 5-9: A class diagram indicates inheritance with a hollow arrowhead.

Activity Diagrams

An *activity diagram* represents work flows for activities. They include several kinds of symbols connected with arrows to show the direction of the work flow. Table 5-4 summarizes the symbols.

TABLE 5-4: Activity Diagram Symbols

SYMBOL	REPRESENTS
Rounded rectangle	An action or task
Diamond	A decision
Thick bar	The start or end of concurrent activities
Black circle	The start
Circled black circle	The end

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Figure 5-10 shows a simple activity diagram for baking cookies.

The first thick bar starts three parallel activities: Start oven, mix dry ingredients, and mix wet ingredients. If you have assistant cookie chefs (perhaps your children, if you have any), those steps can all proceed at the same time in parallel.

When the three parallel activities all are done, the work flow resumes after the second thick bar. The next step is to combine all the ingredients.

A test then checks the batter's consistency. If the batter is too sticky, you add more flour and recheck the consistency. You repeat that loop until the batter has the right consistency.

When the batter is just right, you roll out the cookies, wait until the oven is ready (if it isn't already), and bake the cookies for eight minutes.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

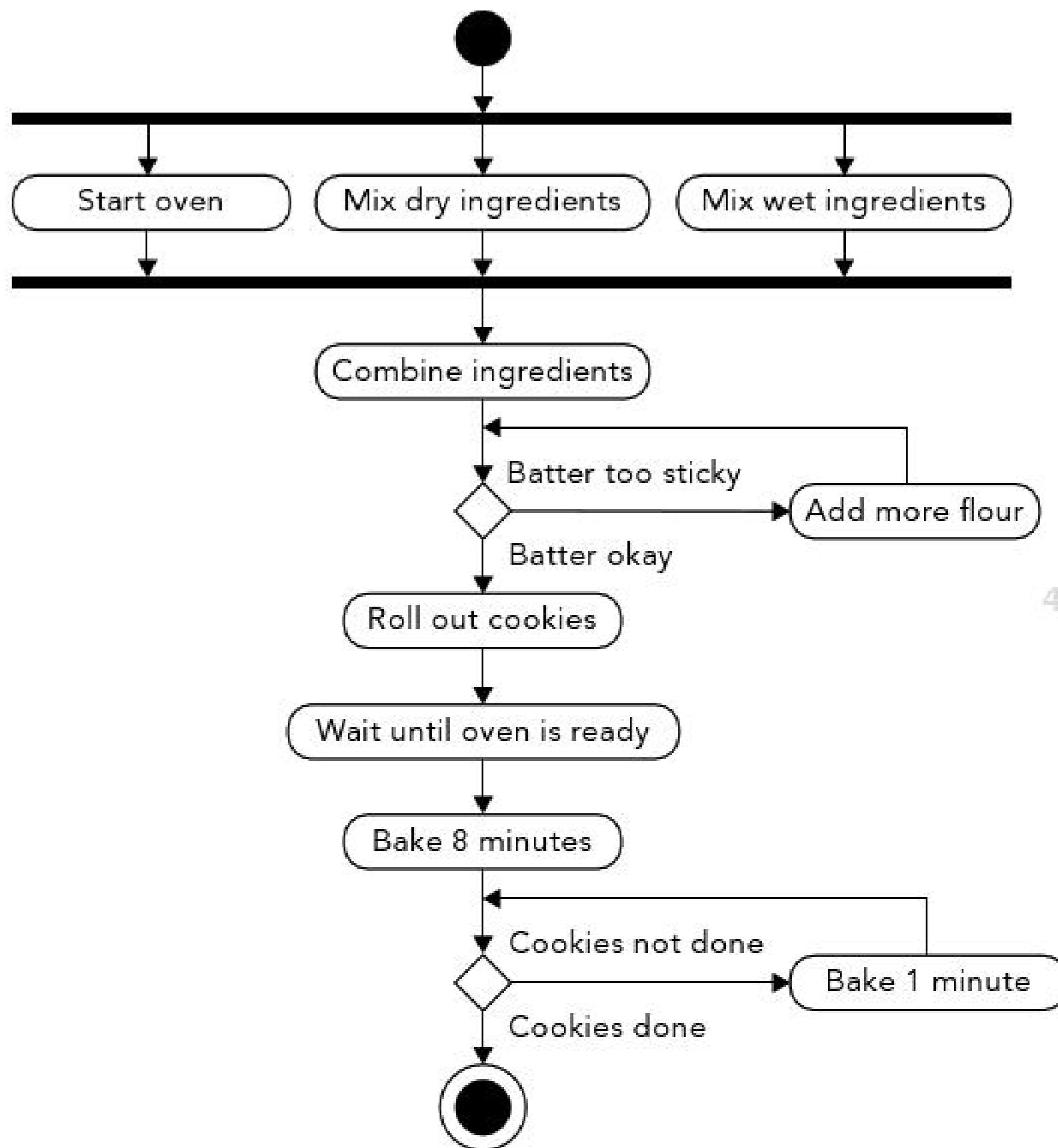


FIGURE 5-10: An activity diagram is a bit like a flowchart showing how work flows.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

After eight minutes, you check the cookies. If the cookies aren't done, you bake them for one more minute. You continue checking and baking for one more minute as long as the cookies are not done.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

When the cookies are done, you enter the stopping state indicated by the circled black circle.

Use Case Diagram

A *use case diagram* represents a user's interaction with the system. Use case diagrams show stick figures representing actors (someone or something that performs a task) connected to tasks represented by ellipses.

To provide more detail, you can use arrows to join subtasks to tasks. Use the annotation `<<include>>` to mean the task includes the subtask. (It can't take place without the subtask.)

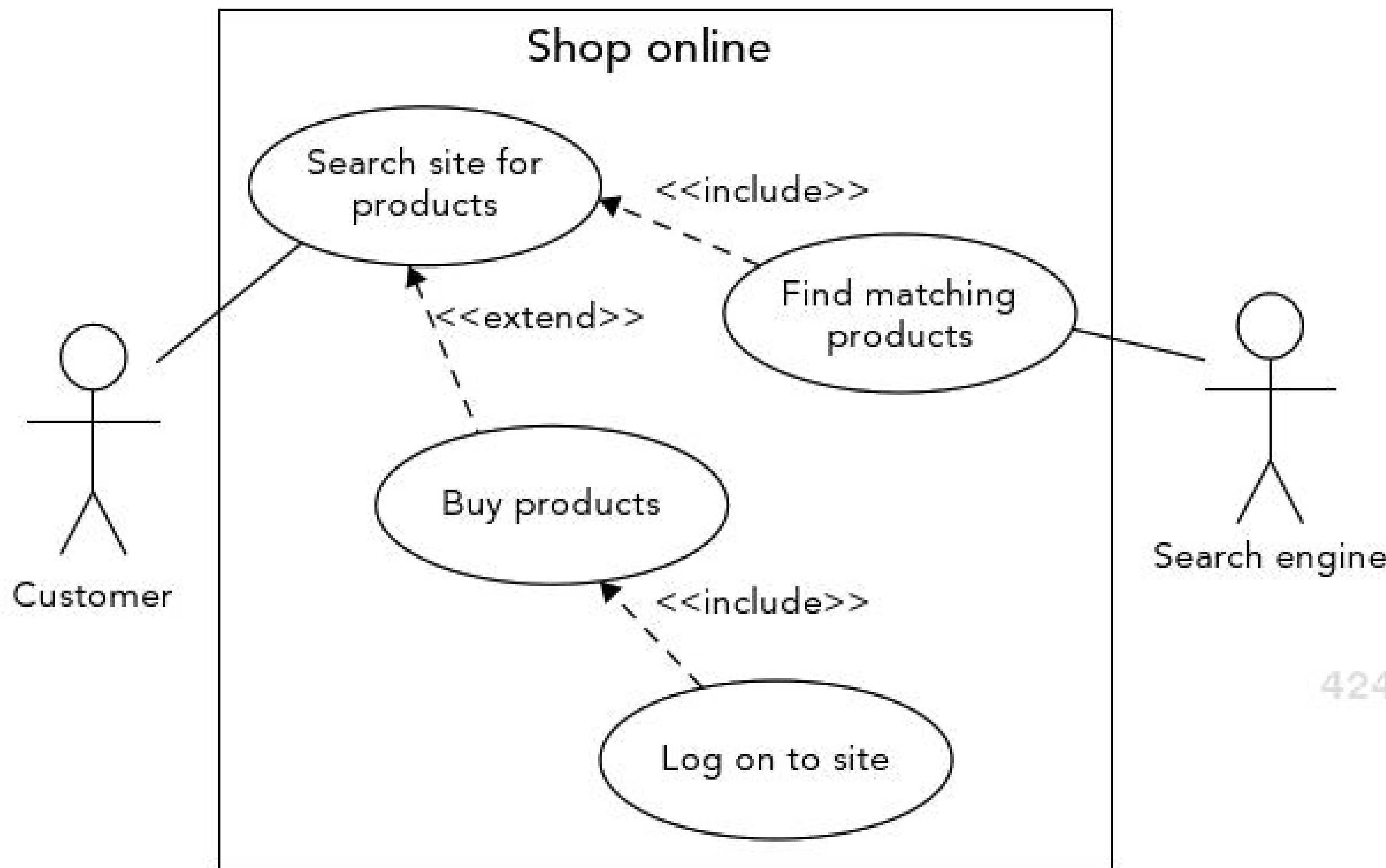
If a subtask might occur only under some circumstances, connect it to the main task and add the annotation `<<extend>>`. If you like, you can add a note indicating when the extension occurs. (Usually both `<<include>>` and `<<extend>>` arrows are dashed.)

Figure 5-11 shows a simple online shopping use case diagram. The customer actor performs the "Search site for products" activity. If he finds something he likes, he also performs the "Buy products" extension. To buy products, the customer must log in to the site, so the "Buy products" activity includes the "Log on to site" activity.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary



424f7c283cced86ffd1b25db1e0b8e1b
ebrary

FIGURE 5-11: A use case diagram shows actors and the tasks they perform (possibly with subtasks and extensions).

The website's search engine also participates in the "Search site for products" activity. When the customer starts a search, the engine performs the "Find matching products" activity. The "Search" activity cannot work without the "Find" activity, so the "Find" activity is included in the "Search" activity.

State Machine Diagram

A *state machine diagram* shows the states through which an object passes in response to various events. States are represented by rounded rectangles. Arrows indicate transitions from one state to another. Sometimes annotations on the arrows indicate what causes a transition.

A black circle represents the starting state and a circled black circle indicates the stopping state.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Figure 5-12 shows a simple state machine diagram for a program that reads a floating point number (as in –17.32) followed by the Enter key.

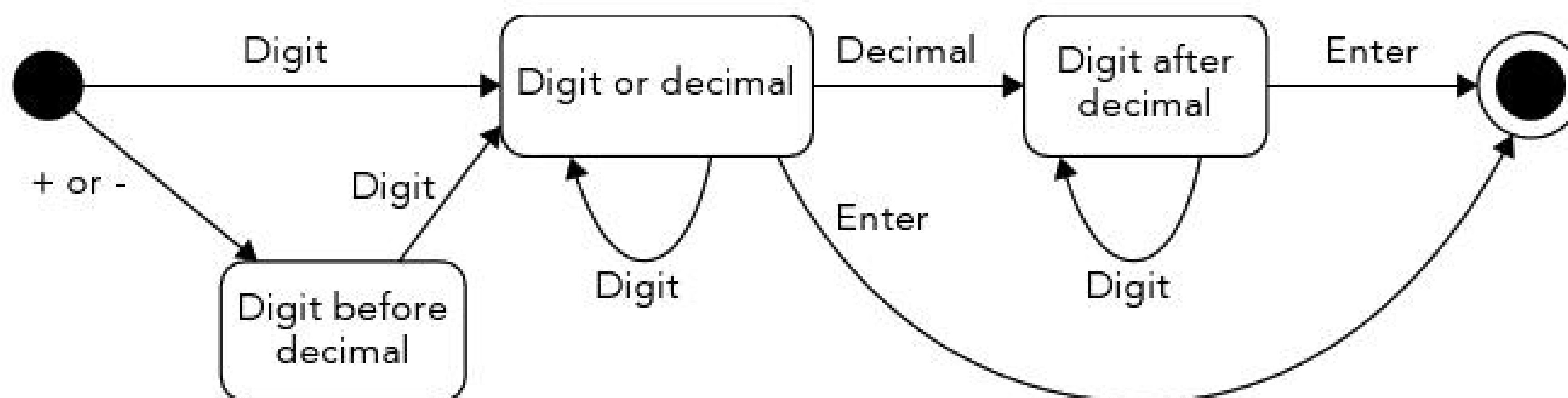


FIGURE 5-12: This state machine diagram represents reading a floating point number.

The program starts and can read a digit, +, or –. (If it reads any other character, the machine fails and the program would need to take some action, such as displaying an error message.) If it reads a +, or –, the machine moves to the state "Digit before decimal."

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

From that state, the user must enter a digit, at which point the machine moves into state "Digit or decimal." The machine also reaches this state if the user initially enters a digit instead of a +, or –.

Now if the user enters another digit, the machine remains in the "Digit or decimal" state. When the user enters a decimal point, it moves to the "Digit after decimal" state. If the user presses the Enter key, the machine moves to its stopping state. (That happens if the user enters a whole number such as 37.)

The machine remains in the "Digit after decimal" state as long as the user types a digit. When the user presses the Enter key, the machine moves to its stopping state.

Interaction Diagrams

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Interaction diagrams are a subset of activity diagrams. They include sequence diagrams, communication diagrams, timing diagrams, and interaction overview diagrams. The following sections provide brief descriptions of these kinds of diagrams and give a few simple examples.

Sequence Diagram

A *sequence diagram* shows how objects collaborate in a particular scenario. It represents the collaboration as a sequence of messages.

Objects participating in the collaboration are represented as rectangles or sometimes as stick figures for actors. They are labeled with a name or class. If the label includes both a name and class, they are separated by a colon.

Below each of the participants is a vertical dashed line called a *lifeline*. The lifeline basically represents the participant sitting there waiting for something to happen.

An *execution specification* (called an *execution* or informally an *activation*) represents a participant doing something. In the diagram, these are represented as gray or white rectangles drawn on top of the lifeline. You can draw overlapping rectangles to represent overlapping executions.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Labeled arrows with solid arrowheads represent synchronous messages. Arrows with open arrowheads represent asynchronous messages. Finally, dashed arrows with open arrowheads represent return messages sent in reply to a calling message.

Figure 5-13 shows a customer, a clerk, and the `Movie` class interacting to print a ticket for a movie. The customer walks up to the ticket window and requests the movie from the clerk. The clerk uses a computer to ask the `Movie` class whether tickets are available for the desired show. The `Movie` class responds.

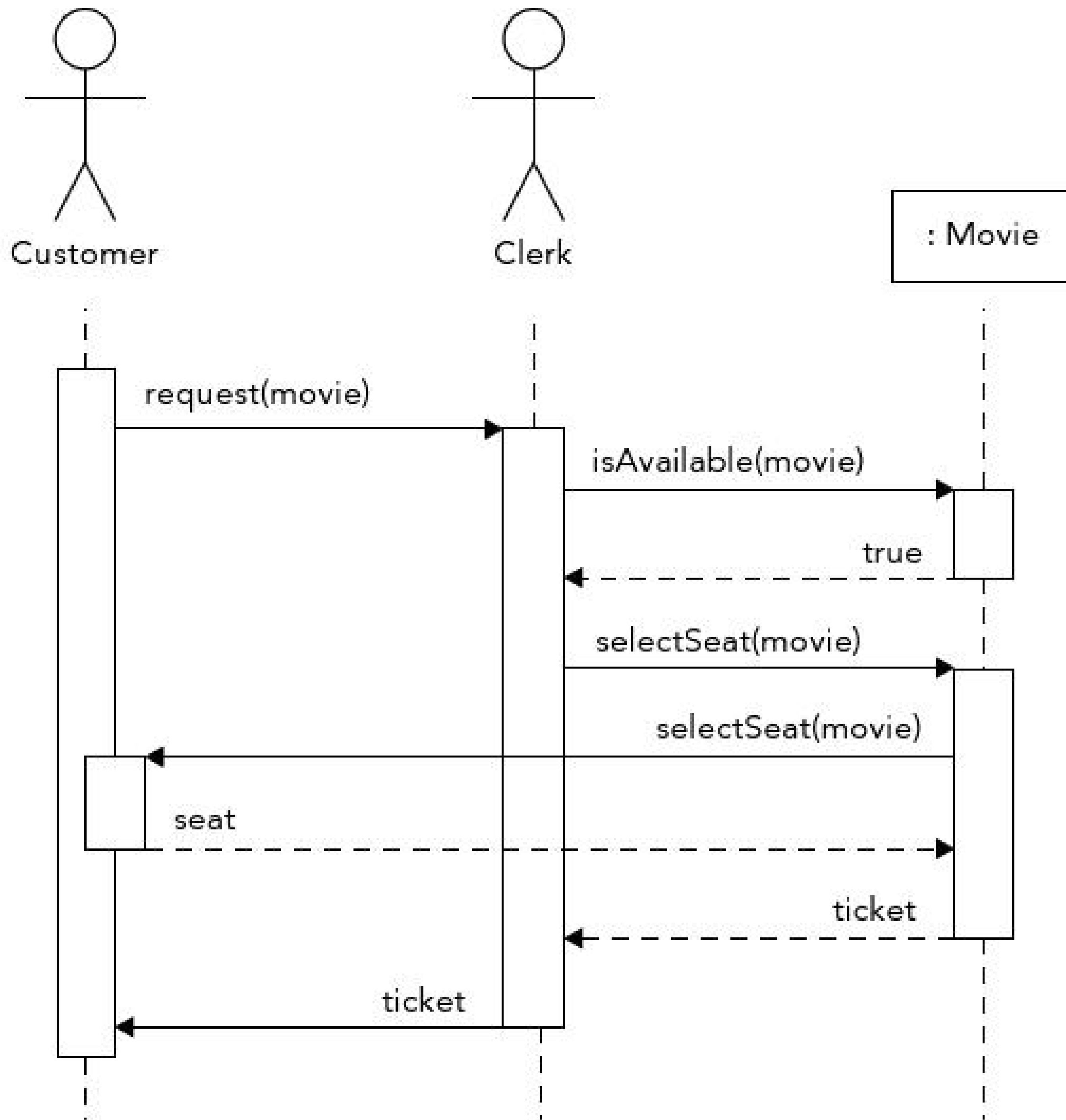
Notice that the `Movie` class's response is asynchronous. The class fires off a response and doesn't wait for any kind of reply. Instead it goes back to twiddling its electronic thumbs, waiting for some other request.

If the class's response is `false`, the interaction ends. (This scenario covers only the customer successfully buying a ticket.) If the response is `true`, control returns to the clerk, who uses the computer to ask the `Movie` class to select a seat. This causes another execution to run on the `Movie` class's lifeline.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary



424f7c283cced86ffd1b25db1e0b8e1b
ebrary

FIGURE 5-13: A sequence diagram shows the timing of messages between collaborating objects.

The `Movie` class in turn asks the customer to pick a seat from those that are available. The customer is still waiting for the initial request to finish, so this is an overlapping execution for the customer.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

After the customer picks a seat, the `Movie` class issues a ticket to the clerk. The clerk then prints the ticket and hands it to the customer.

The point of this diagram is to show the interactions that occur between the participants and the order in which they occur. If you think the diagram is confusing, feel free to add some text describing the process.

Communication Diagram

Like a sequence diagram, a *communication diagram* shows communication among objects during some sort of collaboration. The difference is the sequence diagram focuses on the sequence of messages, but the communication diagram focuses more on the objects involved in the collaboration.

The diagram uses lines to connect objects that collaborate during an interaction. Labeled arrows indicate messages between objects. The messages are numbered that so you can follow the sequence of messages.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Figure 5-14 shows a communication diagram for the movie ticket buying-scenario that was shown in Figure 5-13.

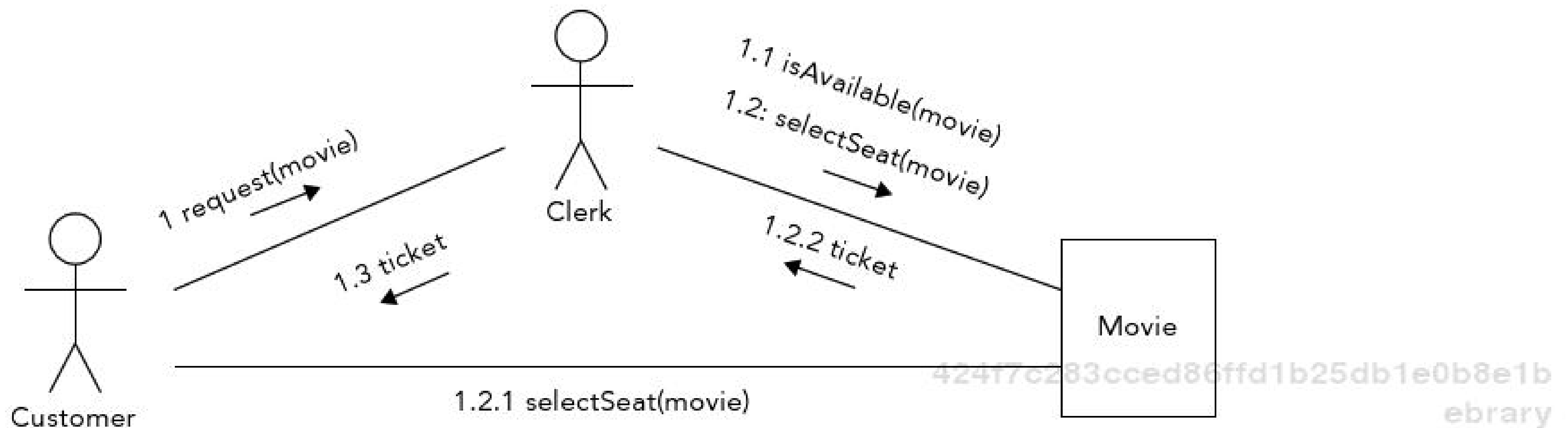


FIGURE 5-14: A communication diagram emphasizes the objects participating in a collaboration.

Following is the sequence of messages in Figure 5-14:

1: The customer asks the clerk for a movie ticket.

1.1: The clerk asks the `Movie` class if a seat is available.

1.2: The clerk asks the `Movie` class to select a seat.

1.2.1: The `Movie` class asks the user to pick a seat.

1.2.2: The `Movie` class sends the clerk a ticket for the selected seat.

1.3: The clerk prints the ticket and hands it to the customer.

The exact timing of the messages and some of the details (such as return messages) are not represented well in the communication diagram. Those details are better represented by a sequence diagram.

Timing Diagram

A *timing diagram* shows one or more objects' changes in state over time. A timing diagram looks a lot like a sequence diagram turned sideways, so time increases from left to right. These diagrams can be useful for giving a sense of how long different parts of a scenario will take.

More elaborate versions of the timing diagram show multiple participants stacked above each other with arrows showing how messages pass between the participants.

Interaction Overview Diagram

An *interaction overview diagram* is basically an activity diagram where the nodes can be frames that contain other kinds of diagrams. Those nodes can contain sequence, communication, timing, and other interaction overview diagrams. This lets you show more detail for nodes that represent complicated tasks.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

SUMMARY

High-level design sets the stage for later software development. It deals with the grand decisions such as:

- What hardware platform will you use?
- What type of database will you use?
- What other systems will interact with this one?
- What reports can you make the users define so you don't have to do all the work?

After you settle these and other high-level questions, the stage is set for development. However, you're still not quite ready to start slapping together code to implement the features described in the requirements. Before you start churning out code, you need to create low-level designs to flesh out the classes, modules, interfaces, and other pieces of the application that you identified during high-level design. The low-level design will give you a detailed picture of exactly what code you need to write so you can begin programming.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The next chapter covers low-level design. It explains how you can refine the database design to ensure the database is robust and flexible. It also describes the kinds of information you need to add to the high-level design before you can start putting 0s and 1s together to make the final program.

EXERCISES

1. What's the difference between a component-based architecture and a service-oriented architecture?
2. Suppose you're building a phone application that lets you play tic-tac-toe against a simple computer opponent. It will display high scores stored on the phone, not in an external database. Which architectures would be most appropriate and why?
3. Repeat question 2 for a chess program running on a desktop, laptop, or tablet computer.
4. Repeat question 3 assuming the chess program lets two users play against each other over an Internet connection.
5. What kinds of reports would the game programs described in Exercises 2, 3, and 4 require?
6. What kind of database structure and maintenance should the ClassyDraw application use?
7. What kind of configuration information should the ClassyDraw application use?
8. Draw a state machine diagram to let a program read floating point numbers in scientific notation as in + 37 or –12.3e + 17 (which means $-12.3 \times 10^{+17}$). Allow both E and e for the exponent symbol.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary
424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

WHAT YOU LEARNED IN THIS CHAPTER

- High-level design is the first step in breaking an application into pieces that are small enough to implement.
- Decoupling tasks allows different teams to work on them simultaneously.
- Some of the things you should specify in a high-level design include:
    - Security (operating system, application, data, network, and physical)
    - Operating system (Windows, iOS, or Linux)
    - Hardware platform (desktop, laptop, tablet, phone, or mainframe)
    - Other hardware (networks, printers, programmable signs, pagers, audio, or video)
    - User interface style (navigational techniques, menus, screens, or forms)
    - Internal interfaces
    - External interfaces
    - Architecture (monolithic, client-server, multitier, component-based, service-oriented, data-centric, event driven, rule-based, or distributed)
    - Reports (application usage, customer purchases, inventory, work schedules, productivity, or ad hoc)
    - Other outputs (printouts, web pages, data files, images, audio, video, e-mail, or text messages)
    - Database (database platform, major tables and their relationships, auditing, user access, maintenance, backup, and data warehousing)
    - Top-level classes (`Customer`, `Employee`, and `Order`)
    - Configuration data (algorithm parameters, due dates, expiration dates, and durations)
    - Data flows
    - Training
- UML diagrams lets you specify the objects in the system (including external agents such as users and external systems) and how they interact.
- The main categories of UML diagrams are structure diagrams and behavior diagrams (which includes the subcategory interaction diagrams).

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

6

Low-Level Design

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

We try to solve the problem by rushing through the design process so that enough time is left at the end of the project to uncover the errors that were made because we rushed through the design process.

—Glenford Myers

WHAT YOU WILL LEARN IN THIS CHAPTER:

- How to use generalization and refinement to build inheritance hierarchies
- Warning signs of bad inheritance hierarchies
- How to use composition to build new classes without inheritance
- How normalization protects databases from anomalies
- Rules for first, second, and third normal forms

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

High-level design paints an application's structure in broad strokes. It identifies the system's general environment (hardware, operating system, network, and so on) and architecture (such as monolithic, client/server, and service-oriented). It identifies the system's major components such as reporting modules, databases, and top-level classes. It should also sketch out how the pieces of the system will interact.

Low-level design fills in some of the gaps to provide extra detail that's necessary before developers can start writing code. It gives more specific guidance for how the parts of the system will work and how they will work together. It refines the definitions of the database, the major classes, and the internal and external interfaces.

High-level design focuses on *what*. Low-level design begins to focus on *how*.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

As an analogy, if you were building a highway system, high-level design would determine what cities (and perhaps what parts of those cities) would be connected by highways. The low-level design would indicate exactly where the highways would be placed, where the ramps would be, and what elementary schools would be surrounded by four-lane traffic circles.

The border between high-level and low-level design is often rather fuzzy. Typically, after a piece of the system is added to the high-level design, team members continue working on that piece to develop its low-level design. Particularly on a large project, some people will be working on high-level designs while others work on low-level designs. Developers may even start implementing parts of the system that have been adequately defined.

In a way, you can describe low-level design as high-level design for micro-managers. You extend the high-level design by providing more and more detail until everything is specified precisely enough to start implementation.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

However, refining a high-level design isn't necessarily easy. You may know generally what you need in the database (customer data and stuff), but unless you refine that knowledge into a good detailed database design, you may run into all sorts of problems later. The data may become inconsistent, the program might lose critical information, and finding data may be slow. Different database designs can make the difference between finding the data you need in seconds, hours, or not at all.

The following sections describe some of the most important concepts you should keep in mind during low-level design. They explain how to refine an object model to identify the application's classes, how to use stepwise refinement to provide additional detail for a task, and how to design a database that is flexible and robust.

OO DESIGN

The high-level design should have identified the major types of classes that the application will use. Now it's time to refine that design to identify the specific classes that program will need. The new classes should include definitions of the properties, methods, and events they will provide for the application to use.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

A QUICK OO PRIMER

In object-oriented (OO) development, classes define the general properties and behaviors for a set of objects. An *instance* of a class is an object with the class's type.

For example, you could define an `Author` class to represent authors. An instance of the class might represent the specific author William Shakespeare. After you define the `Author` class, you could create any number of instances of that class to represent different authors.

Classes define three main items: properties, methods, and events.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

A *property* is something that helps define an object. For example, the `Author` class might have `FirstName` and `LastName` properties to identify the specific author an instance represents. It might have other properties such as `DateOfBirth`, `DateOfDeath`, and `WrittenWorks`.

A *method* is a piece of code that makes an object do something. The `Author` class might have a `Search` method that searches an object's `WrittenWorks` values for a work that contains a certain word. It might also have methods to print a formatted list of the author's works, or to search online for places to buy one of the author's works.

An *event* is something that occurs to tell the program that something interesting has happened. An object *raises* an event when appropriate to let the program take some action. For example, an `Author` object might raise a `Birthday` event to tell the program that today is the author's birthday. (That would be hard for Shakespeare because no one knows exactly he was born.) When the program creates an `Author` object, that object would raise the `Birthday` event if it were the author's birthday. It would also raise that event if the object already existed and the clock ticked over past midnight so that it became the author's birthday.

After you design a class, you can use it like a cookie cutter to make as many instances of the class as you like. Each instance has the same properties, methods, and events; although, the properties can have different values in different instances.

Object-oriented development involves lots of other details, but this should be enough to get you through the following discussion. If you want more information about object-oriented programming, look for a book on the subject, either in general or for your favorite programming language.

Also look for books on design patterns. An object-oriented *design pattern* is an arrangement of classes that performs some common and useful task. For example, the model-view-controller (MVC) pattern breaks a user interface interaction into three pieces: a model object that represents some data, view objects that display a view of the data to the user, and controller objects that control the model, possibly allowing the user to manipulate the data. Design patterns can be useful in designing the classes that make up an application, but they're outside the scope of this book.

The following sections explain how you can define the classes that an application will use.

Identifying Classes

The previous chapter tells you that you should identify the main classes that the application will use, but it doesn't tell you how to do that. One way to pick classes is to look for nouns in a description of the application's features.

For example, suppose you're writing an application called FreeWheeler Automatic Driver (FAD) that automatically drives cars. Now consider the sentence, "The program drives the car to the selected destination." That sentence contains three nouns: program, car, and destination.

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The program probably doesn't need to directly manipulate itself, so it's unlikely that you'll need a `Program` class. It will almost certainly need to work with cars and destinations, so you probably do need `Car` and `Destination` classes.

When you're studying possible classes, think about what sorts of information the class needs (properties), what sorts of things it needs to do (methods), and whether it needs to notify the program of changing circumstances (events). For this example, the `Car` class is going to be fully loaded, providing all sorts of properties (such as `CurrentSpeed`, `CurrentDirection`, and `FuelLevel`), methods (such as `Accelerate`, `Decelerate`, `ActivateTurnSignal`, and `HonkHorn`), and events (such as `DriverPressedStart`, `FuelLevelLow`, and `CollisionImminent`).

The `Destination` class is probably a lot simpler because it basically just represents a specific location. In fact, it may be that the application needs only a single instance of this class to record the current destination.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Making only a single instance of a class is a warning sign that perhaps the class isn't necessary. The fact that the `Destination` class doesn't do anything or change on its own (so it doesn't provide methods or events) is another indication that you might not need that class. In this example, you could store the destination information in a couple variables holding latitude and longitude.

Note that the class definitions depend heavily on how you will use the objects. For example, you could define a `Passenger` class to represent people riding in the car. A passenger has all sorts of interesting information such as `Name`, `Address`, `Age`, and `CreditScore`. However, the FreeWheeler program doesn't need to know any of that information. It might not even need to know if the car contains any passengers. (Although it probably needs to have a driver, at least until automated cars become so good they can travel on their own.)

Building Inheritance Hierarchies

After you define the application's main classes, you need to add more detail to represent variations on those classes. For example, FreeWheeler is going to need a `Car` class to represent the vehicle it's driving, but different vehicles have different characteristics. A 106-horsepower Toyota Yaris handles differently than a 460-horsepower Chevrolet Corvette. It would be bad if the program told the Yaris to pull out in front of a speeding tractor trailer, assuming it could go from 0 to 60 miles per hour in 3.7 seconds.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You can capture the differences between related classes by *deriving* a *child class* from a *parent class*. In this example, you might derive the `Yaris` and `Corvette` child classes from the `Car` parent class.

Child classes automatically inherit the properties, methods, and events defined by the parent class. For example, the `Car` class might define methods such as `SetParkingBrake`, `TurnLeft`, and `DeployDragChute`. Because `Corvette` inherits from the `Car` class, a `Corvette` object automatically knows how to perform those methods.

This is one important way object-oriented programming languages achieve code reuse. You write code once in the parent class and any child classes use that same code without you rewriting it.

The fact that `Corvette` inherits from `Car` also means that a `Corvette` is a kind of `Car`. Intuitively, that makes sense. In real life, a Corvette is a car, so it should do anything that any other car can do.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Because an instance of a child class also belongs to the parent class, the program should be able to treat the object as if it were of the parent class if that would be helpful. In this example, that means a program should be able to treat a `Corvette` object as either a `Corvette` or as a more generic `Car`. For instance, the program could create an array of `Car` objects and fill it with instances of the `Corvette`, `Yaris`, `VolkswagenBeatle`, or `DeLorean` classes. The program should be able to treat all those objects as if they were `Cars` without knowing their true classes. The capability to treat objects as if they were actually from a different class is called *polymorphism*.

You can derive multiple classes from a single parent class. For example, you could derive `Corvette`, `Edsel`, and `Pinto` all from the `Car` class.

Conversely, most object-oriented programming languages do not allow *multiple inheritance*, so a class can have at most a single parent class. Because classes can have at most one parent but any number of children, the relationships between classes form a tree-like *inheritance hierarchy*.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

There are a lot of ways you can modify basic inheritance relationships. For example, a child class can add properties, methods, and events (which together are called *members*) that are not available in the parent class. A child class can also replace a parent class member with a new version.

In some languages the child class can even define a new version of a member that applies when the program refers to an object by using the child class but not when it refers to it with a variable that has the parent class's type. For example, you might give the `Car` class a `ParallelPark` method that carefully backs the car into a parking space. The `Corvette` class might define a new version that locks up the brakes and slides the car into the space sideways as if James Bond were driving. Now if the program defines a variable of type `Car` that refers to a `Corvette` object and invokes its `ParallelPark` method, you get the first version. If the program defines a second variable of type `Corvette` that refers to the same object and invokes its `ParallelPark` method, you get the second version.

The details of how you define classes, build inheritance hierarchies, and add or modify their members depend on the language you use, so those things aren't covered in this book. Before moving on to other topics, however, you should know about the two main ways for building inheritance hierarchies: refinement and generalization.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Refinement

Refinement is the process of breaking a parent class into multiple subclasses to capture some difference between objects in the class. When I derived the `Corvette`, `Edsel`, and `Pinto` classes from the `Car` class, that was refinement.

One danger to refinement is *overrefinement*, which happens when you refine a class hierarchy unnecessarily, making too many classes that make programming more complicated and confusing. People are naturally good at categorizing objects. It takes only a few seconds of thought to break cars into the classes shown in Figure 6-1. The open arrowheads point from child classes to their parent classes.

With a bit more work, you can grow this hierarchy until it is truly enormous. There are a couple hundred models of car on the roads in the United States alone. You could refine most of those models with different options such as different engine sizes, radios, speakers, alloy wheels, spoilers, and seat warmers. You could add still more subclasses to represent different colors.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

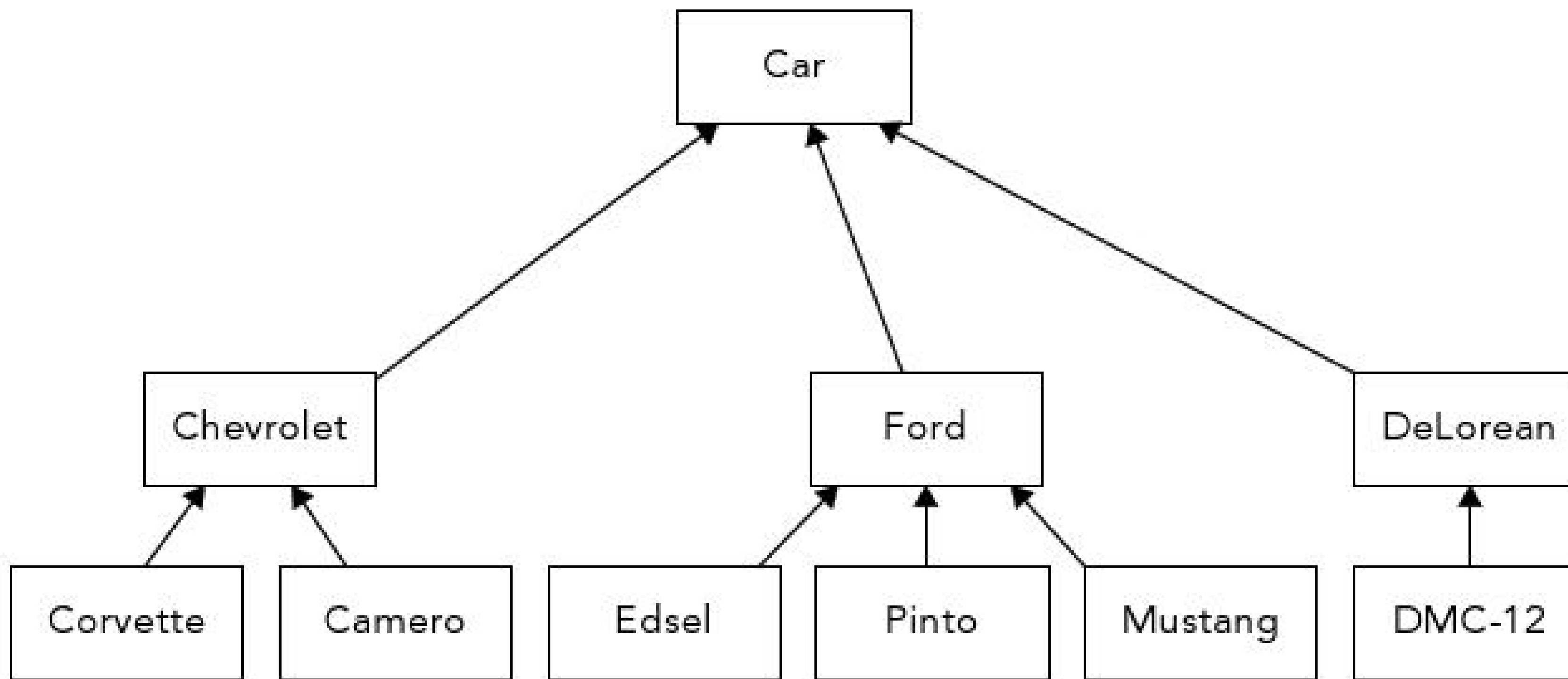


FIGURE 6-1: People are naturally good at building inheritance hierarchies.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The resulting hierarchy would contain many thousands (possibly millions) of classes. Obviously, a hierarchy that large wouldn't be useful. There's no way you could write enough code to actually use each of the classes, and if you're not going to use a class, why build it?

There are two main problems here. First, the classes are capturing data that isn't relevant to the application. The FreeWheeler application doesn't care what color a car is or whether it has a CD changer. It only cares about the car's driving characteristics: mileage, maximum acceleration, turn radius, and so forth. The hierarchy in Figure 6-1 doesn't capture any of that information.

RISKY REFINEMENT

Even if the program cares about certain differences between objects, that doesn't mean those differences would make a good inheritance hierarchy. For example, suppose you're writing a car sales application. Customers often want to shop for cars first by make, then by model, and then by option packages and other features. In that case, the customer's search strategy looks a lot like Figure 6-1.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Unfortunately, if you use those values to build the inheritance hierarchy, you get a monstrously huge hierarchy. Even though the program cares a lot about those differences, they're better handled as properties rather than subclassing. It's easy enough for a program to search a database for specific property values such as make or model without storing the data in a hierarchical format.

The second problem with this hierarchy is that the differences between cars could easily be represented by properties instead of by different classes. The differences identified so far actually are just different values for the same properties. For example, Chevrolet, Ford, and DeLorean are all just different values for a `Make` property. You could eliminate that whole level of the hierarchy by simply adding a `Make` property to the `Car` class.

Similarly, a car's model (Corvette, Edsel, and Mustang) is just a name for a specific type of car. You may have some expectations based on the name (you probably think a Corvette is faster than a Pinto), but to the FreeWheeler program, those are just labels.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You can avoid these kinds of hierarchy problems if you focus on behavioral differences between the different kinds of objects instead of looking at differences in properties.

For example, what are the behavioral differences between a Corvette and a Pinto? The Corvette accelerates quicker, but both cars *can* accelerate, just at different rates. They still have the same acceleration behavior, so you can represent that difference as an `Acceleration` property in the `Car` class.

For an example where there is a behavioral difference, consider transmission type. To accelerate a car with automatic transmission to freeway speeds, you simply stomp on the gas pedal until the car is going fast enough. Bringing a manual transmission car up to speed is much more complicated, requiring you to use the gas pedal, the clutch, and the gear shift. Both kinds of vehicles accelerate, but the details about how they do it are different.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In object-oriented terms, the `Car` class might have an `Accelerate` method that makes the car accelerate. The `Automatic` and `Manual` subclasses would provide different implementations of the `Accelerate` method that handle the appropriate details.

Figure 6-2 shows a revised inheritance hierarchy. The first section under a class's name lists its properties (just `Acceleration` in this example). A subclass does not repeat items that it inherits without modification from its parent class. In this example, the `Automatic` and `Manual` classes inherit the `Acceleration` property.

The second section below a class's name shows methods (`Accelerate` in this example). The method is italicized in the `Car` class to indicate that it is not implemented there and must be overridden in the child classes.

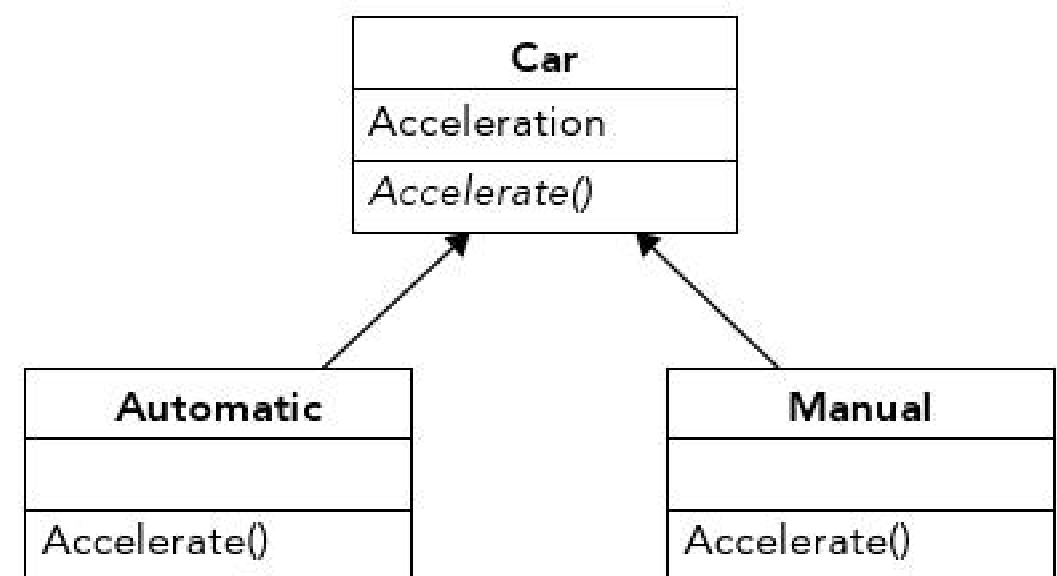


FIGURE 6-2: This hierarchy focuses on behavioral differences between classes.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Generalization

Refinement starts with a single class and creates child classes to represent differences between objects. Generalization does the opposite: It starts with several classes and creates a parent for them to represent common features.

For example, consider the ClassyDraw application in the examples in Chapter 4, "Requirement Gathering," and Chapter 5, "High-Level Design." This program is a drawing application somewhat similar to MS Paint, except it allows you to manipulate drawn objects. It enables you to select an object, drag it into a new position, stretch it, move it to the top or bottom of the stacking order, delete it, copy and paste it, and so forth.

The program represents drawn objects as (you guessed it) objects, so it needs classes such as `Rectangle`, `Ellipse`, `Polygon`, `Text`, `Line`, `Star`, and `Hypotrochoid`.

These classes draw different shapes, but they also have a lot in common. They all let you click their object to select it, move the object to the top or bottom of the drawing order, move the object, and so forth.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Because all those objects share these features, it makes sense to create a parent class that defines them. The program can build a big array or list to hold all the drawing objects represented by the parent class and then use polymorphism to invoke the common methods as necessary.

For a concrete example, suppose the user clicks part of a drawing to select a drawn object. Classes such as `Rectangle` and `Ellipse` use different techniques to decide whether you clicked their objects, but they both need a method to do that. You could call this method `ObjectIsAt` and make it return true if the object is at a specific clicked location. The parent class, which I'll call `Drawable`, can define the `ObjectIsAt` method. The child classes would then provide their own implementations.

Figure 6-3 shows the drawing class inheritance hierarchy.

Just as you can go overboard with refinement to build an inheritance hierarchy containing thousands of car classes, you can also get carried away with generalization. For example, suppose you're building a pet store inventory application. You define a `Customer` class and an `Employee` class. They share some properties such as `Name`, `Address`, and `ZodiacSign`, so you generalize them by making a `Person` class to hold the common properties.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

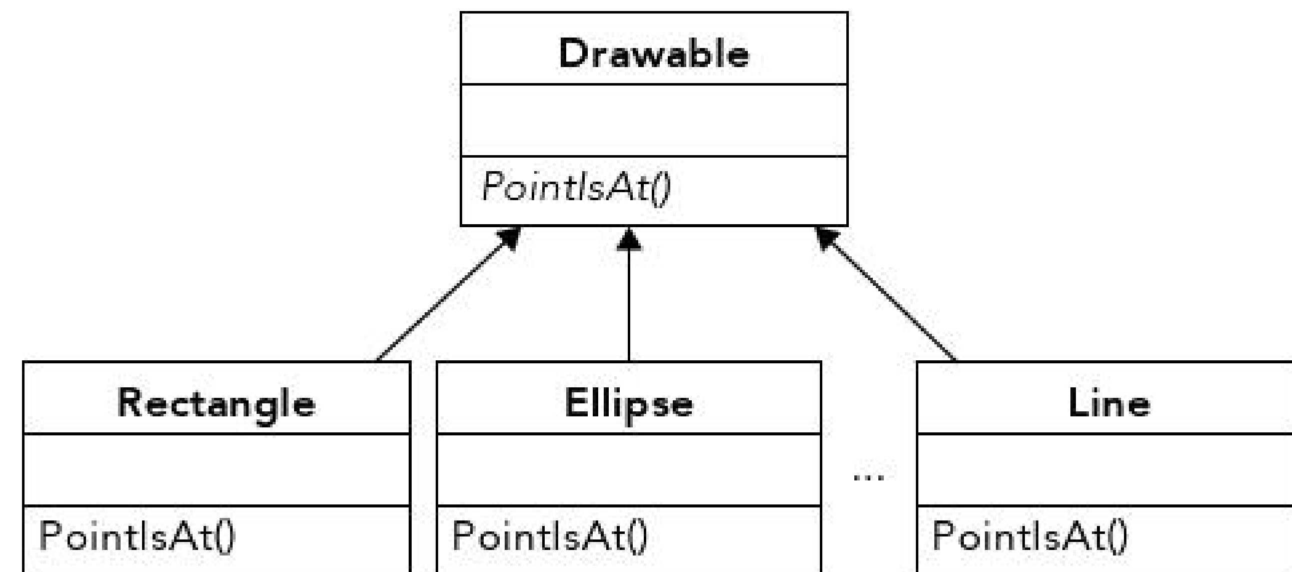


FIGURE 6-3: Generalization creates the Drawable parent class.

Next, you define various pet classes such as `Dog`, `Cat`, `Gerbil`, and `Capybara`. You generalize them to make a `Pet` class.

In a fit of inspiration (possibly assisted by whatever you were drinking), you realize that people and pets are all animals! So you make an `Animal` class to be a parent class for `Person` and `Pet`. They can even share some properties such as `Name`.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Logically, this makes sense. People and pets really are animals (as long as your pet store doesn't sell pet rocks or stuffed toys). However, it's unlikely that the program will ever take advantage of this fact. It's hard to imagine the program building an array or list containing both employees and birds and then treating them in a uniform way. In all likelihood, the program will treat people and pets in different ways, so they don't need to be merged into a single inheritance hierarchy.

Hierarchy Warning Signs

The following list gives some questions you can ask yourself when trying to decide if you have an effective inheritance hierarchy.

- Is it tall and thin? In general, tall, thin inheritance hierarchies are more confusing than shorter ones. Tall hierarchies make it hard for developers to remember which class to use under different circumstances. How tall an inheritance hierarchy can be depends on your application, but if it contains more than three or four levels, you should make sure you really need them all.
- Do you have a huge number of classes? Suppose your car sales application needs to track make, model, year, color, engine, wheel size, and motorized cup holders. If you try to use

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

classes to represent every possible combination, you'll get a combinatorial explosion and thousands of classes. If you have more than a dozen or so classes, see if you can replace some with simple properties.

- Does a class have only a single subclass? If so, then you can probably remove it and move whatever it was trying to represent into the subclass.
- If there a class at the bottom of the hierarchy that is never instantiated? If the `Car` hierarchy has a `HalfTrack` class and the program never makes an instance of that class, then you probably don't need the `HalfTrack` class.
- Do the classes all make common sense? If the `Car` hierarchy contains a `Helicopter` class, there's probably something wrong. Either the class doesn't belong there or you should rename some classes so things make sense. (Perhaps you need a `Vehicle` class?)
- Do classes represent differences in a property's value rather than in behavior or the presence of properties? A simple sales program might not need separate classes to represent notebooks and three-hole punches because they're both simple products that you sell one at a time. You might want a separate class for more expensive objects like computers because they might have a `Warranty` property that notebooks and hole punches probably don't have.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Object Composition

Inheritance is one way you can reuse code. A child class inherits all of the code defined by its parent class, so you don't need to write it again. Another way to reuse code is *object composition*, a technique that uses existing classes to build more complex classes.

For example, suppose you define a `Person` class that has `FirstName`, `LastName`, `Address`, and `Phone` properties. Now you want to make a `Company` class that should include information about a contact person.

You could make the `Company` class inherit from the `Person` class so it would inherit the `FirstName`, `LastName`, `Address`, and `Phone` properties. That would give you places to store the contact person's information, but it doesn't make intuitive sense. A company is not a kind of person (despite certain Supreme Court rulings), so `Company` should not inherit from `Person`.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

A better approach is to give the `Company` class a new property of type `Person` called `ContactPerson`. Now the `Company` class gets the benefit of the code defined by the `Person` class without the illogic and possible confusion of inheriting from `Person`.

This approach also lets you place more than one `Person` object inside the `Company` class. For example, if you decide the `Company` class also needs to store information about a billing contact and a shipping contact, you can add more `Person` objects to the class. You couldn't do that with inheritance.

DATABASE DESIGN

There are many different kinds of databases that you can use to build an application. For example, specialized kinds of databases store hierarchical data, documents, graphs and networks, key/value pairs, and objects. However, the most popular kind of databases are relational databases.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

DATABASE RANKINGS

To see the top database engines ranked by popularity, go to `db-engines.com/en/ranking`. It's a pretty interesting list.

Relational databases are simple, easy to use, and provide a good set of tools for searching, combining data from different tables, sorting results, and otherwise rearranging data.

Like object-oriented design, database design is too big a topic to squeeze into a tiny portion of this book. However, there is room here to cover a few of the most important concepts of database design. You can find a book on database design for more complete information. (For example, see my book *Beginning Database Design Solutions*, Wrox, 2008.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The following section briefly explains what a relational database is. The sections after that explain the first three forms of database normalization and why they are important.

Relational Databases

Before you learn about database normalization, you need to at least know the basics of relational databases.

A *relational database* stores related data in *tables*. Each table holds *records* that contain pieces of data that are related. Sometimes records are called *tuples* to emphasize that they contain a set of related values.

The pieces of data in each record are called *fields*. Each field has a name and a data type. All the values in different records for a particular field have that data type.

Figure 6-4 shows a small `Customer` table holding five records. The table's fields are `CustomerId`, `FirstName`, `LastName`, `Street`, `City`, `State`, and `Zip`. Because the representation shown in Figure 6-4 lays out the data in rows and columns, records are often called *rows* and fields are often called *columns*.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

CustomerId	FirstName	LastName	Street	City	State	Zip
1028	Veronica	Jenson	176 Bradley Ave	Abend	AZ	87351
2918	Kirk	Wood	61 Beech St	Bugsville	CT	04514
7910	Lila	Rowe	8391 Cedar Ct	Cobblestone	SC	35245
3198	Deirdre	Lemon	2819 Dent Dr	Dove	DE	29183
5002	Alicia	Hayes	298 Elf Ln	Eagle	CO	83726

FIGURE 6-4: A table's records are often called rows and its fields are often called columns.

The "relational" part of the term "relational database" comes from relationships defined between the database's tables. For example, consider the `Orders` table shown in Figure 6-5. The `Customers` table's `CustomerId` field and the `Orders` table's `CustomerId` field form a relationship between the two tables. To find a particular customer's orders, you can look up that customer's `CustomerId` in the `Customers` table in Figure 6-4, and then find the corresponding `Orders` records.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

CustomerId	OrderId	DateOrdered	DateFilled	DateShipped
1028	1298	4/1/2015	4/4/2015	4/4/2015
2918	1982	4/1/2015	4/3/2015	4/4/2015
3198	2917	4/2/2015	4/7/2015	4/9/2015
1028	9201	4/5/2015	4/6/2015	4/9/2015
1028	3010	4/9/2015	4/13/2015	4/14/2015

FIGURE 6-5: The Customers table's CustomerId column provides a link to the Orders table's CustomerID column.

One particularly useful kind of relationship is a foreign key relationship. A ***foreign key*** is a set of one or more fields in one table with values that uniquely define a record in another table.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

For example, in the `Orders` table shown in Figure 6-5, the `CustomerId` field uniquely identifies a record in the `Customers` table. In other words, it tells you which customer placed the order. There may be multiple records in the `Orders` table with the same `CustomerId` (a single customer can place multiple orders), but there can be only one record in the `Customers` table that has a particular `CustomerId` value.

The table containing the foreign key is often called the ***child table***, and the table that contains the uniquely identified record is often called the ***parent table***. In this example, the `Orders` table is the child table, and the `Customers` table is the parent table.

LOOKUP TABLES

A *lookup table* is a table that contains values just to use as foreign keys.

For example, you could make a `States` table that lists the states that are allowed by the application. If your company has customers only in New England, the table might contain the values Maine, New Hampshire, Vermont, Massachusetts, Connecticut, and Rhode Island.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The `Customers` table would be a child table connected to the `States` table with a foreign key. That would prevent a user from adding a new customer in a state that wasn't allowed.

In addition to validating user inputs, lookup tables allow the users to configure the application. If you let users modify the `States` table, they can add new records when they decide to work with customers in new states.

Building a relational database is easy, but unless you design the database properly, you may encounter unexpected problems. Those problems may be that:

- Duplicate data can waste space and make updating values slow.
- You may be unable to delete one piece of data without also deleting another unrelated piece of data.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

- An otherwise unnecessary piece of data may need to exist so that you can represent some other data.
- The database may not allow multiple values when you need them.

The database-speak euphemism for these kinds of problems is *anomalies*.

Database normalization is a process of rearranging a database to put it into a standard (normal) form that prevents these kinds of anomalies. There are seven levels of database normalization that deal with increasingly obscure kinds of anomalies. The following sections describe the first three levels of normalization, which handle the worst kinds of database problems.

First Normal Form

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

First normal form (*1NF*) basically says the table can be placed meaningfully in a relational database. It means the table has a sensible, down-to-earth structure like the kind your grandma used to make.

Relational database products tend to enforce most of the 1NF rules automatically, so if you don't do anything too weird, your database will be in 1NF with little extra work.

The official requirements for a table to be in 1NF are:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.
3. Each column must have a single data type.
4. No two rows can contain identical values.
5. Each column must contain a single value.
6. Columns cannot contain repeating groups.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

To see how you might be tricked into breaking these rules, suppose you're a weapons instructor at a fantasy adventure camp. You teach kids how to whack each other safely with foam swords and the like. Now consider the signup sheet shown in Table 6-1.

TABLE 6-1: Weapons Training Signup Sheet

NAME	WEAPON	WEAPON
Shelly Silva	Broadsword	
Louis Christenson	Bow	
Lee Hall	Katana	
Sharon Simmons	Broadsword	Bow
Felipe Vega	Broadsword	Katana
Louis Christenson	Bow	
Kate Ballard	Everything	

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Here campers list their names and weapons for which they want training. You'll call them in for instruction on a first-come-first-served basis.

This signup sheet violates the 1NF rules in several ways.

It violates Rule 1 because it contains two columns named `Weapon`. The idea is that a camper might want help with more than one weapon. That makes sense on a signup sheet but won't work in a relational database.

It violates Rule 2 because the order of the rows indicates the order in which the campers signed up and the order in which you'll tutor them. In other words, the ordering of the rows is important. (The order of the columns might also be important if you assume the first `Weapon` column holds the camper's primary weapon.)

It violates Rule 3 because Kate Ballard didn't enter the name of a weapon in the first weapon column. Ideally, that column's data type would be `Weapon` and campers would just enter a weapon's name, not a general comment such as "Everything."

It violates Rule 4 because Louis Christenson signed up twice for tutoring with the bow. (I guess he wants to get *really* good with the bow.)

The signup sheet doesn't violate Rule 5, but that's mostly due to luck. There's nothing (except common sense) to stop campers from entering multiple weapons in each `Weapon` column, and that would violate Rule 5.

Here's how you can put this signup sheet into 1NF.

Rule 1—The signup sheet has two columns named `Weapon`. You can fix that by changing their names to `Weapon1` and `Weapon2`. (That violates Rule 6, but we'll fix that later.)

Rule 2—The order of the rows in the signup sheet determines the order in which you'll call campers for their tutorials, so the ordering of rows is important. To fix this problem, add a new field that stores the ordering data explicitly. One way to do that would be to add an `Order` field, as shown in Table 6-2.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TABLE 6-2: Ordered Signup Sheet

ORDER	NAME	WEAPON1	WEAPON2
1	Shelly Silva	Broadsword	
2	Louis Christenson	Bow	
3	Lee Hall	Katana	
4	Sharon Simmons	Broadsword	Bow
5	Felipe Vega	Broadsword	Katana
6	Louis Christenson	Bow	
7	Kate Ballard	Everything	

An alternative that might be more useful would be to add a `Time` field instead of an `Order` field, as shown in Table 6-3. That preserves the original ordering and gives extra information that the campers can use to schedule their days.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TABLE 6-3: Signup Sheet with Times

TIME	NAME	WEAPON1	WEAPON2
9:00	Shelly Silva	Broadsword	
9:30	Louis Christenson	Bow	
10:00	Lee Hall	Katana	
10:30	Sharon Simmons	Broadsword	Bow
11:00	Felipe Vega	Broadsword	Katana
11:30	Louis Christenson	Bow	
12:00	Kate Ballard	Everything	

Rule 3—In Table 6-3, the `Weapon1` column holds two kinds of values: the name of a weapon or "Everything" (for Kate Ballard).

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Depending on the application, there are several approaches you could take to fix this kind of problem. You could split a column into two columns, each containing a single data type. Alternatively, you could move the data into separate tables linked to the original record by a key.

In this example, I'll replace the value "Everything" with multiple records that list all the possible weapon values. The result is shown in Table 6-4.

TABLE 6-4: Signup Sheet with Explicitly Listed Weapons

TIME	NAME	WEAPON1	WEAPON2
9:00	Shelly Silva	Broadsword	
9:30	Louis Christenson	Bow	
10:00	Lee Hall	Katana	
10:30	Sharon Simmons	Broadsword	Bow
11:00	Felipe Vega	Broadsword	Katana
11:30	Louis Christenson	Bow	
12:00	Kate Ballard	Broadsword	
12:00	Kate Ballard	Bow	
12:00	Kate Ballard	Katana	

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Rule 4—The current design doesn't contain any duplicate rows, so it satisfies Rule 4.

Rule 5—Right now each column contains a single value, so the current design satisfies Rule 5. (The original signup sheet would have broken this rule if it had used a single `Weapons` column instead of using two separate columns and people had written in lists of the weapons they wanted to study.)

Rule 6—This rule says a table cannot contain repeating groups. That means you can't have two columns that represent the same thing. This means a bit more than two columns don't have the same *data type*. Tables often have multiple columns with the same data types but with different meanings. For example, the `Camper` table might have `HomePhone` and `CellPhone` fields. Both of them would hold phone numbers, but they represent different *kinds* of phone numbers.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In the current design, the `Weapon1` and `Weapon2` columns hold the same type and kind of data, so they form a repeating group.

ROTTEN REPETITION

In general, adding a number to field names to differentiate them is a bad idea. If the program doesn't need to differentiate between the two values, then adding a number to their names just creates a repeating group.

The only time this makes sense is if the two fields contain similar items that truly have different meanings to the application. For example, suppose a space shuttle requires two pilots: one to be the primary pilot and one to be the backup in case the primary pilot is abducted by aliens. In that case, you could name the fields that store their names `Pilot1` and `Pilot2` because there really is a difference between them.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Usually in cases like this, you can give the fields more descriptive names such as `Pilot` and `Copilot`.

Another way to look at this is to ask yourself whether the record "Sharon Simmons, Broadsword, Bow" and the rearranged record "Sharon Simmons, Bow, Broadsword" would have the same meaning. If the two have the same meaning even if you switch the values of the two fields, then those fields form a repeating group.

The way to fix this problem is to pull the repeated data out into a new table. Use fields in the original table to link to the new one. Figure 6-6 shows the new design. Here the `Tutorials` and `TutorialWeapons` tables are linked by their `Time` fields.

Tutorials	
Time	**Name**
9:00	Shelly Silva
9:30	Louis Christenson
10:00	Lee Hall
10:30	Sharon Simmons
11:00	Felipe Vega
11:30	Louis Christenson
12:00	Kate Ballard

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TutorialWeapns	
Time	**Weapon**
9:00	Broadsword
9:30	Bow
10:00	Katana
10:30	Broadsword
10:30	Bow
11:00	Broadsword
11:00	Katana
11:30	Bow
12:00	Broadsword
12:00	Bow
12:00	Katana

FIGURE 6-6: This design is in first 1NF. Lines connect related records.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Second Normal Form

A table is in *second normal form* (2NF) if it satisfies these rules:

1. It is in 1NF.
2. All non-key fields depend on all key fields.

Without getting two technical, a *key* is a set of one or more fields that uniquely identifies a record. Any table in 1NF must have a key because 1NF Rule 4 says, "No two rows can contain identical values." That means there must be a way to pick fields to guarantee uniqueness, even if the key must include every field.

For an example of a table that is not in 2NF, suppose you want to schedule games for campers at the fantasy adventure camp. Table 6-5 lists the scheduled games.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TABLE 6-5: Camp Games Schedule

TIME	GAME	DURATION	MAXIMUMPLAYERS
1:00	*Goblin Launch*	60 mins	8
1:00	*Water Wizzards*	120 mins	6
2:00	*Panic at the Picnic*	90 mins	12
2:00	*Goblin Launch*	60 mins	8
3:00	*Capture the Castle*	120 mins	100
3:00	*Water Wizzards*	120 mins	6
4:00	*Middle Earth Hold'em Poker*	90 mins	10
5:00	*Capture the Castle*	120 mins	100

The table's primary key is `Time+Game`. It cannot have two instances of the same game at the same time (because you don't have enough equipment or counselors), so the combination of `Time+Game` uniquely identifies the rows.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You should quickly review the 1NF rules and convince yourself that this table is in 1NF. In case you haven't memorized them yet, the 1NF rules are:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.
3. Each column must have a single data type.
4. No two rows can contain identical values.
5. Each column must contain a single value.
6. Columns cannot contain repeating groups.

Even though this table is in 1NF, it suffers from the following anomalies:

- **Update anomalies**—If you modify the `Duration` or `MaximumPlayers` value in one row, other rows containing the same game will be out of sync.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

- **Deletion anomalies**—Suppose you want to cancel the *Middle Earth Hold'em Poker* game at 4:00, so you delete that record. Then you've lost all the information about that game. You no longer know that it takes 90 minutes and has a maximum of 10 players.
- **Insertion anomalies**—You cannot add information about a new game without scheduling it for play. For example, suppose *Banshee Bingo* takes 45 minutes and has a maximum of 30 players. You can't add that information to the database without scheduling a game.

The problem with this table is that it's trying to do too much. It's trying to store information about both games (duration and maximum players) and the schedule.

The reason it breaks the 2NF rules is that some non-key fields do not depend on *all* the key fields. Recall that this table's key fields are `Time` and `Game`. A game's duration and maximum number of players depends only on the `Game` and not on the `Time`. For example, *Water Wizzards* lasts for 120 minutes whether you play at 1:00, 4:00, or midnight.

To fix this table, move the data that doesn't depend on the *entire* key into a new table. Use the key fields that the data does depend on to link to the original table.

Figure 6-7 shows the new design. Here the `ScheduledGames` table holds schedule information and the `Games` table holds information specific to the games.

ScheduledGames	
Time	**Game**
1:00	Goblin Launch
1:00	Water Wizzards
2:00	Panic at the Picnic
2:00	Goblin Launch
3:00	Capture the Castle
3:00	Water Wizzards
4:00	Middle Earth Hold'em Poker
5:00	Capture the Castle

Games		
Game	**Duration**	**MaximumPlayers**
Goblin Launch	60 min	8
Water Wizzards	120 min	6
Panic at the Picnic	90 min	12
Capture the Castle	120 min	100
Middle Earth Hold'em Poker	90 min	10
Banshee Bingo	45 min	30

FIGURE 6-7: Moving the data that doesn't depend on *all* the table's key fields puts this table in 2NF.

Third Normal Form

A table is in *third normal form* (*3NF*) if:

1. It is in 2NF.
2. It contains no transitive dependencies.

A *transitive dependency* is when a non-key field's value depends on another non-key field's value.

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

For example, suppose the fantasy adventure camp has a library. (So campers have something to read after they get injured playing the games.) Posted in the library is the following list of the counselors' favorite books, as shown in Table 6-6.

TABLE 6-6: Counselors' Favorite Books

COUNSELOR	FAVORITEBOOK	AUTHOR	PAGES
Becky	*Dealing with Dragons*	Patricia Wrede	240
Charlotte	*The Last Dragonslayer*	Jasper Fforde	306
J.C.	*Gil's All Fright Diner*	A. Lee Martinez	288
Jon	*The Last Dragonslayer*	Jasper Fforde	306
Luke	*The Color of Magic*	Terry Pratchett	288
Noah	*Dealing with Dragons*	Patricia Wrede	240
Rod	*Equal Rites*	Terry Pratchett	272
Wendy	*The Lord of the Rings Trilogy*	J.R.R. Tolkein	1178

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

This table's key is the `Counselor` field.

If you run through the 1NF rules, you'll see that this table is in 1NF.

The table has only a single key field, so a non-key field cannot depend on only *some* of the key fields. That means the table is also in 2NF.

When posted on the wall of the library, this list is fine. Inside a database, however, it suffers from the following anomalies:

- **Update anomalies**—If you change the `Pages` value for Becky's row (*Dealing with Dragons*), it will be inconsistent with Noah's row (also *Dealing with Dragons*). Also if Luke changes his favorite book to *Majestrum: A Tale of Hengis Hapthorn*, the table loses the data it has about *The Color of Magic*.
- **Deletion anomalies**—If J.C. quits being a counselor to become a professional wrestler and you remove his record from the table, you lose the information about *Gil's All Fright Diner*.
- **Insertion anomalies**—You cannot add information about a new book unless it's someone's favorite. Conversely, you can't add information about a person unless he declares a favorite book.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The problem is that some non-key fields depend on other non-key fields. In this example, the `Author` and `Pages` fields depend on the `FavoriteBook` field. For example, any record with `FavoriteBook` *The Last Dragonslayer* has `Author` Jasper Fforde and `Pages` 306 no matter whose favorite it is.

DIAGNOSING DEPENDENCIES

A major hint that there is a transitive dependency in this table is that there are lots of duplicate values in different columns. Another way to think about this is that there are "tuples" of data (`FavoriteBook+Author+Pages`) that go together.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You can fix this problem by keeping only enough information to identify the dependent data and moving the rest of those fields into a new table. In this example, you would keep the `FavoriteBook` field in the original table and move its dependent values `Author` and `Pages` into a new table. Figure 6-8 shows the new design.

CounselorFavorites	
Counselor	**FavoriteBook**
Becky	Dealing with Dragons
Charlotte	The Last Dragonslayer
J.C.	Gil's All Fright Diner
Jon	The Last Dragonslayer
Luke	The Color of Magic
Noah	Dealing with Dragons
Rod	Equal Rites
Wendy	The Lord of the Rings Trilogy

BookInfo		
Book	**Author**	**Pages**
Dealing with Dragons	Patricia Wrede	240
The Last Dragonslayer	Jasper Fforde	306
Gil's All Fright Diner	A. Lee Martinez	288
The Color of Magic	Terry Pratchett	288
Equal Rites	Terry Pratchett	272
The Lord of the Rings Trilogy	J.R.R. Tolkein	1178

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

FIGURE 6-8: Moving non-key fields that depend on other non-key fields into a separate table puts this table in 3NF.

Higher Levels of Normalization

Higher levels of normalization include Boyce-Codd normal form (BCNF), fourth normal form (4NF), fifth normal form (5NF), and Domain/Key Normal Form (DKNF). Some of these later levels of normalization are fairly technical and confusing, so I won't cover them here. See a book on database design for details.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Many database designs stop at 3NF because it handles most kinds of database anomalies without a huge amount of effort. In fact, with a little practice, you can design database tables in 3NF from the beginning, so you don't need to spend several steps normalizing them.

More complete levels of normalization can also lead to confusing database designs that may make using the database harder and less intuitive, possibly giving rise to extra bugs and sometimes reduced performance.

One particular compromise that is often useful is to intentionally leave some data denormalized for performance reasons. A classic example is in ZIP codes. ZIP codes and street addresses are related, so if you know a street address, you can look up the corresponding ZIP code. For example, the ZIP code for 1 Main St., Boston, MA is 02129-3786.

Ideally, normalization would tell you to store only the street address and then use it to look up the ZIP code as needed. Unfortunately, these relationships aren't as simple as, "All Main St. addresses in Boston have the ZIP code 02129-3786." ZIP codes depend on which part of the street contains the address and sometimes even which side of the street the address is on. That means you can't build a table to perform a simple lookup.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You could build a much more complicated table to find an address's ZIP code, perhaps with some confusing code. Or you might use some sort of web service provided by the United States Postal Service.

Usually, however, developers just include the ZIP code as a separate field in the address. That means there's a lot of "unnecessary" duplication, but it doesn't take up much extra room and it makes looking up addresses much easier.

LOADS OF CODES

Addresses and postal codes are also related outside of the United States. For example, the postal code for 1 Main St., Dungiven, Londonderry England is BT47 4PG, and the postal code for 1 Main St., Vancouver, BC, Canada is V6A 3Y5. You can use various postal websites to look up codes for different addresses in different countries.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In theory, you could look up the postal codes for any address. In practice, it's a lot easier to just include them in the address data.

SUMMARY

Low-level design fills in some of the gaps left by high-level design to provide extra guidance to developers before they start writing code. It provides the level of detail necessary for programmers to start writing code, or at least for them to start building classes and to finish defining interfaces. Low-level design moves the high-level focus from *what* to a lower level focus on *how*.

Like most of the topics covered in this book, low-level design is a huge subject. There's no way to cover every possible approach to low-level design in a single chapter. However, this chapter does provide an introduction to two important facets of low-level design: object-oriented design and database design.

Object-oriented design determines what classes the application uses. Database design determines what tables the database contains and how they are related. Object-oriented design and database design aren't all you need to do to ensure success, but poor designs almost always lead to failure.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The boundary between high-level and low-level design is rather arbitrary. Low-level design tasks are similar to high-level design tasks but with a greater level of detail. In fact, the same kinds of tasks can slip into the next step in software engineering: development.

The next chapter provides an introduction to software development. It explains some general methods you can use to organize development. It also describes a few useful techniques you can use to reduce the number of bugs that are introduced during development.

EXERCISES

1. **Consider the `ClassyDraw` classes `Line`, `Rectangle`, `Ellipse`, `Star`, and `Text`. What properties do these classes all share? What properties do they not share? Are there any properties shared by some classes and not others? Where should the shared and nonshared properties be implemented?**

2. **Draw an inheritance diagram showing the properties you identified for Exercise 1. (Create parent classes as needed, and don't forget the `Drawable` class at the top.)**

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

3. The following list gives the properties of several business-oriented classes.

   - `Customer—Name, Phone, Address, BillingAddress, CustomerId`
   - `Hourly—Name, Phone, Address, EmployeeId, HourlyRate`
   - `Manager—Name, Phone, Address, EmployeeId, Office, Salary, Boss, Employees`
   - `Salaried—Name, Phone, Address, EmployeeId, Office, Salary, Boss`
   - `Supplier—Name, Phone, Address, Products, SupplierId`
   - `VicePresident—Name, Phone, Address, EmployeeId, Office, Salary, Managers`

   Assuming a `Supplier` is someone who supplies products for your business, draw an inheritance diagram showing the relationships among these classes. (Hint: Add extra classes if necessary.)

4. How would the inheritance hierarchy you drew for Exercise 3 change if you decide to add the `Boss` property to the `Hourly` class?

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

5. How would the inheritance hierarchy you drew for Exercise 3 change if `Supplier` represents a business instead of a person?

6. Suppose your company has many managerial types such as department manager, project manager, and division manager. You also have multiple levels of vice president, some of whom report to other manager types. How could you combine the `Salaried`, `Manager`, and `VicePresident` types you used in Exercise 3? Draw the new inheritance hierarchy.

7. If a table includes a ZIP code with every address, what 1NF, 2NF, and 3NF rules does the table break?

8. What data anomalies can result from including postal codes in address data? How bad are they? How can you mitigate the problems?

9. In the United States Postal Service's ZIP+4 system, ZIP codes can include 4 extra digits as in 20500-0002. Suppose you store address data with a single `Zip` field that has room for 10 characters. Some addresses include only a 5-digit ZIP code and others include a ZIP+4 code. Does that violate any of the 1NF, 2NF, or 3NF rules? Should you do anything about it?

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

10. Do telephone area codes face issues similar to those involving ZIP codes?

11. Suppose you're writing an application to record times for dragon boat races and consider the table shown in Figure 6-9. Assume the table's key is `Heat`. What 1NF, 2NF, and 3NF rules does this design violate?

12. How could you fix the table shown in Figure 6-9?

Distance	Heat	Time	Team	Team	Winner	Time	Time
500	1	9:00	Buddhist Temple	Wicked Wind	Buddhist Temple	2:55.372	2:57.391
500	2	9:20	Rainbow Energy	Rising Typhoon	Rising Typhoon	3:10.201	3:01.791
1000	3	9:40	Math Dragons	Supermarines	Math Dragons	5:52.029	6:23.552
1000	4	10:00	Flux Lake Tritons	Elf Power	Elf Power	6:08.480	6:59.717

FIGURE 6-9: This table records dragon boat race results.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

WHAT YOU LEARNED IN THIS CHAPTER

- A class defines the properties, methods, and events provided by instances of the class.
- Nouns in the project description make good candidates for classes.
- Inheritance provides code reuse.
- Polymorphism lets a program treat an object as if it had a parent class's type.
- In refinement, you add details to a general class to define subclasses.
- In generalization, you extract common features from two or more classes to define a parent class.
- Inheritance hierarchy warning signs include:
  - The hierarchy is tall and thin.
  - The hierarchy contains a large number of classes.
  - A class has a single subclass.
  - A class at the bottom of the hierarchy is never instantiated.
  - The classes don't make common sense.
  - Classes represent differences in property values, not different properties themselves or different behaviors.
- Composition provides code reuse. It also lets you include multiple copies of a type of object inside a class, something inheritance doesn't do.
- Relational databases contain tables that hold records (or rows). The records in a table all have the same fields (or columns).
- A foreign key forms a relationship between the values in a parent table and the values in a child table. The child table's fields must contain values that are present in the parent table.
- A lookup table is a foreign key parent table that simply defines values that are allowed in other tables.
- Normalization protects a database from data anomalies.
- 1NF rules:
  1. Each column must have a unique name.
  2. The order of the rows and columns doesn't matter.
  3. Each column must have a single data type.
  4. No two rows can contain identical values.
  5. Each column must contain a single value.
  6. Columns cannot contain repeating groups.

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

- 2NF rules:
    1. It is in 1NF.
    2. All non-key fields depend on all key fields.
- 3NF rules:
    1. It is in 2NF.
    2. It contains no transitive dependencies. (No non-key fields depend on other non-key fields.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

7

Development

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

A good programmer is someone who always looks both ways before crossing a one-way street.

—Doug Linder

Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.

—Martin Golding

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Tools that are useful to programmers
- How to decide which algorithms are better than others
- How to use top-down design to turn designs into code
- Programming tips that can make code easier to debug and maintain

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

To many programmers, development is the heart of software engineering. It's where fingers hit the keyboard and churn out the actual program code of the system. Without development, there is no application.

As is the case with other stages of software development, the edges of development are a bit blurry. Low-level design may identify the classes that a program will need, but it may not spell out every method that the classes must provide and it might provide few details about how those methods work. That means the development stage must still include some design work as developers figure out how to build the classes.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Similarly, the next stage of software engineering, testing, often begins before development is completely finished. In fact, it's best to test software early and often. It's widely known that bugs are easiest to find and fix if they're detected soon after they're created, so if possible you should test every method you write as soon as it's finished (sometimes even before it's finished).

Most developers write programs because they like to write code. (I know I do. For me, solving a tricky programming problem is like solving a difficult Sudoku puzzle. I get a great feeling of satisfaction from crunching a bunch of numbers and having a beautiful fractal or a three-dimensional game pop out.) Over the years, programmers have collectively spent a huge amount of time programming, fixing bugs in their code, and thinking of ways to avoid similar bugs in the future. That has generated a huge number of books about programming style and techniques for avoiding, detecting, and fixing bugs.

This chapter provides an introduction to some of the techniques that I've found most useful over the years. It begins by describing some tools and general problem-solving approaches that you can use to turn the description of a method into code. It then explains some specific techniques that you can use to make your code easier to debug and maintain.

If you're not a programmer, for example, if you're a project manager or a customer, you may not need to memorize every one of these rules and apply them to your daily life. However, it's still worth your time to read them so that you'll know what's involved in writing good code (and so you'll understand what the programmers are complaining about).

USE THE RIGHT TOOLS

Including overhead (office space, computer hardware, network hardware, Internet service provider, vacation, sick time, a well-stocked soda machine, and so forth), employing a programmer can easily cost more than $100,000 per year. Still I have seen managers refuse to spend a few hundred bucks for proper programming tools. I've seen projects end the year with thousands of dollars left over for hardware expenses, but not a nickel for software tools.

When you're spending $50 per hour on each employee, you don't have to save much of their time to make a little extra expense worthwhile. You don't need to go crazy and spend thousands of dollars to buy everyone a high-end video recording package (unless that's what your business does), but you should spend a little money to make sure your team has all the tools it needs.

The following sections describe some of the development tools that every programmer should have.

Hardware

Few things are as frustrating as trying to write software on inadequate hardware. Programmers need fast computers with lots of memory and disk space. A programmer with an underpowered computer or insufficient memory takes longer to do everything.

Even worse, waiting for slow compilations breaks the programmer's train of thought. To write bug-free (or at least minimally buggy) code, a programmer must stay focused on a method's design until it has been completely written. Breaking the writing process into dozens of chunks separated by several minutes of thumb twiddling (or more likely, trips to the water cooler) breaks the programmer's

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

train of thought, so he needs to re-create an understanding of the code each time. If each new understanding doesn't match the previous ones, the result is far more likely to contain bugs.

THE TORTOISE AND THE SLOTH

I once worked on an application using a slow development environment. I would add the next feature to the code, press the Compile button, and wander off for five or six minutes to wait for the compilation to finish. It was just too frustrating to sit there staring at the screen while the compiler slowly dribbled out periods to tell me that it was working and not dead.

Meanwhile my business partner was stuck on another project but with a similar development cycle. We spent a lot of time in the hallway talking about vacations.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

After about a month of that, I got a new development environment that had a lot fewer features but that was much faster. It let me reproduce everything I had done in the previous month in just two days. As you can probably guess, I never used the other environment again.

Make sure the programmers have all the hardware resources they need to do their jobs quickly and effectively. If that means buying more memory, disk space, or even new computers, do it. It's insane to waste hundreds of hours a year of a programmer's time to save a few hundred dollars. (Although I've known managers who did exactly that. In fact, I've known managers who wouldn't pay for new hardware for their programmers, but who needed the absolute top-of-the line computers for themselves so that they could fill out expense reports and answer e-mail.)

There are two drawbacks to buying the programmers everything they need. First, some programmers will go overboard and buy all sorts of fun toys that they don't actually need. Most programmers don't need a USB controlled NERF rocket launcher or a Darth Vader USB hub. If you have the money, you might let some of those purchases slide in the interests of morale. Otherwise, you might want to check the product SKUs on the purchase requisitions you're signing. (I've known people to try to requisition Dalmatian puppies and cars, mostly as jokes. Those were caught, but I know of one lab that managed to buy a hot tub one piece at a time. They got in a whole lot of… well…hot water.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The second and far more important drawback to giving developers everything they want is that they sometimes forget that their users may not have such nice equipment. I've used applications that were blazingly fast on the developers' computer but that were painfully slow for the users. Modern computers are fast enough and cheap enough that this isn't the problem it used to be in the "old days" two or three years ago, but you should always test applications with hardware that is similar to whatever your end users will be stuck with.

Network

I've known development groups that didn't allow access to external networks. I can understand why that might be necessary if you're designing a new *Minecraft* mod and are worried that foreign

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

hackers will steal your plans and sell them to terrorists, but if it's at all possible, you should allow programmers to have free access to the Internet. Often a quick search can find a solution to a programming problem that would otherwise take hours to solve.

For example, when I'm working on a tricky project, I often use my own websites (`www.csharphelper.com` and `www.vb-helper.com`) to look up specific techniques. My sites are particularly useful to me because they hold solutions to lots of problems I've encountered in the past and because I know more or less what they contain. I also often find solutions on Wikipedia (`www.wikipedia.org`) and I find a lot of mathematical solutions on Wolfram MathWorld (`mathworld.wolfram.com`). And, of course, I often use a search engine to look for other solutions.

You should gently encourage staff members not to spend their whole day playing *Cookie Clicker* or in chat rooms arguing about who is better, Kirk or Picard, but try to provide a fast Internet connection and the freedom to use it.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Development Environment

This is the absolute minimum necessary to make programming possible. It at least includes the compiler or interpreter that translates program code into something the computer can execute.

An *integrated development environment* (*IDE*) such as Eclipse (mostly for Java, although plug-ins let you write in other languages such as C++ or Ruby) or Visual Studio (for Visual C#, Visual Basic, Visual C++, JavaScript, and F#) can also include much more. Depending on the version you have installed, they can include debuggers, code profilers, class visualization tools, auto-completion when typing code, context-sensitive help, team integration tools, and more.

Note that you don't always need the fanciest development environment possible. For example, Visual Studio comes in many different versions, ranging from the free "express" edition designed for individual users, to the "professional" and "ultimate" editions designed for large project teams, which cost a whole lot (MSRP, prices in U.S. dollars). The more expensive versions include tools and resources that are most useful for larger projects so, if you're writing a small application by yourself, you may do just as well with the free express edition.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Similarly, Eclipse comes in a variety of IDEs with a lot of different plug-ins to meet the needs of different kinds of users. For example, Eclipse for Testers is designed for testers. (Well, duh.) If you're not doing a lot of testing, you may want to use a different version.

WHO KNOWS

Most mature development environments include remarkably powerful tools for writing code. They're so effective for a very good reason: the programmers who wrote them know what you need to write programs. In contrast, programmers may not know a lot about court reporting software, medical diagnostics, or cabinet design. However, if programmers know anything, they know what features make development environments effective. There will always be some variation, and you may need to pay extra to get the best features, but there are some amazingly powerful tools out there if you're willing to learn how to use them.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Source Code Control

If your development environment doesn't include source code control, a separate system is essential. Chapter 2, "Before the Beginning," explains that a documentation management system is important for letting you track the many documents that make up a project. Source code control is even more important for program code where changing a single character can reduce a working program to a worthless pile of gibberish.

A good source code management system enables you to go back through past versions of the software and see exactly what changes were made and when. If a program stops working, you can pull out old versions of the code to see which changes broke the program. After you know exactly what changes were made between the last working version and the first broken one, you can figure out which changes caused the bug and you can fix them.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Source code control programs also prevent multiple programmers from tripping over each other as they try to modify the same code at the same time.

Profilers

Profilers let you determine what parts of the program use the most time, memory, files, or other resources. These can save you a huge amount of time when you're trying to tune an application's performance. (I'll say more about this in the section "Defer Optimization" later in this chapter.)

You may not need to buy every programmer a profiler. Typically, a small part of a program's code determines its overall performance, so you usually don't need to study every line's performance extensively. Still it's important to have profilers available when they are needed.

Static Analysis Tools

Profilers monitor a program as it executes to see how it works. Static analysis tools study code without executing it. They tend to focus on the code's style. For example, they can measure how interconnected different pieces of code are or how complex a piece of code is. They can also calculate statistics that may indicate code quality and maintainability such as the number of comments per line of code and average the number of lines of code per method.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Testing Tools

Testing tools, particularly automated tools, can make testing a whole lot faster, easier, and more reliable. I'll talk more about testing tools in the next chapter (which covers testing). For now, just be aware that every programmer must perform at least some testing, so everyone should have access to testing tools.

Source Code Formatters

Some development environments do a better job of formatting code than others. For example, some environments automatically indent source code to show how code is nested in `if-then` statements and loops. That formatting makes code easier to read and understand. That in turn reduces the number of bugs in the code and makes finding and fixing bugs easier.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Other development environments don't provide much in the way of formatting. If you're using that kind of environment, a separate code formatter can standardize indentation, align and reformat comments, break code so it fits on a printout, enforce some code standards, and more.

(Your team will need to decide on the level of code uniformity you want to enforce. Too much standardization can be annoying to developers, but left to their own devices, a few programmers will produce such free-spirited results that their code looks more like an E.E. Cummings poem than professional software.)

Refactoring Tools

The term *refactoring* is programmer-speak for "rearranging code to make it easier to understand, more maintainable, or generally better." Some refactoring tools (which may be built into the IDE) let you do things like easily define new classes or methods, or extract a chunk of code into a new method.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Refactoring tools can be particularly useful if you're managing existing code (as opposed to writing new code).

Training

This is another category where some managers are penny-wise and pound-foolish. Training makes programmers more effective and keeps them happy. A few thousand dollars spent on training can greatly improve performance and help you retain your staff.

Online video training courses and books are often less effective than in-person training, but they're also a lot less expensive and they let you study whenever you have the time. If a $50 book gives you a single new tip, then it's probably worth it.

You do need to be a little selective, however. If you buy too many books, you won't have time to read them all.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

SELECTING ALGORITHMS

After low-level design is mostly complete (and you have all your tools in place), you should have a good sense of what classes you need and the tasks those classes need to perform. The next step is writing the code to perform those tasks.

For more complicated problems, the first step is researching possible algorithms. An *algorithm* is like a recipe for solving a hard programming problem. In the decades since computers were invented, many efficient algorithms have been developed to solve problems such as the following:

- Sorting and arranging pieces of data
- Quickly locating items in databases
- Finding optimal paths through street, power, communication, or other networks
- Designing networks to provide necessary capacity and redundancy to prevent single points of failure

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

- Encrypting and decrypting data
- Picking optimal investment strategies
- Finding least cost construction and production strategies
- Many, many more

For complicated problems like these, the difference between a good algorithm and a bad one can make the difference between finding a good solution in seconds, hours, days, or not at all.

Fortunately, these sorts of algorithms have been extensively studied for years, so you usually don't need to write your own from scratch. You can use the Internet and algorithm books to look for an approach that fits your problem. (For example, see my introductory book *Essential Algorithms: A Practical Approach to Computer Algorithms*, Wiley, 2013.) 424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You'll probably still need to do some work plugging the algorithm into your application, but there's no need for you to reinvent everything from scratch. However, even if you don't need to build an algorithm from the ground up, you should know some of the characteristics that make an algorithm a good choice for you. The following sections describe some of those characteristics.

Effective

Obviously, an algorithm won't do you much good if it doesn't solve your particular problem. An algorithm that finds critical paths through a PERT chart (remember those from Chapter 3, "Project Management"?) won't help you much with calculating the ideal maintenance schedule for a fleet of trucks. You need to pick the right algorithm for the job.

If an algorithm doesn't meet your needs exactly, look for an algorithm that does. If you can find something that only comes close but doesn't *quite* fit your situation, ask yourself whether the algorithm's result is good enough or if you can adjust your requirements a bit to make the available algorithm usable.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

If you can't find an algorithm that fits your problem, and you can't adjust your problem to fit the available algorithms, then you may need to write your own algorithm or modify an existing one. Complicated algorithms often include some of the most highly studied and optimized code you will ever encounter, so modifying them can be difficult. (That difficulty can also make it a fun challenge, but complicated algorithms should probably come with a sticker that says, "Modify at your own risk.")

If you do need to write your own algorithm or modify an existing one, be sure to perform extra testing to make sure it works correctly.

Efficient

The best algorithm in the world won't do you much good if it takes seven years to build the daily production schedule or if it requires the users to have 3 petabytes (1 million gigabytes) of memory on their cell phones. To be useful, an algorithm must satisfy your speed, memory, disk space, and other requirements.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

This is one of the reasons Chapter 4 said requirements must be verifiable. If you don't know ahead of time how quickly the program must find a result, how can you know whether the algorithm you've selected is fast enough?

Note that some algorithms may be efficient enough for one purpose but not for another. For example, suppose you and your friend discover a pirate treasure. Each piece of treasure has a different value, and you want to divide the treasure as equally as possible. (In the algorithm literature, this is called the "partition problem"; although, I like to call it the "booty division problem").

One algorithm for solving this problem is to simply try every possible division of the spoils and see which combination gives you the best result. For example, if there are three pieces of treasure labeled A, B, and C, then there are only eight possible ways to divide the treasure. Table 7-1 shows the possible combinations.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TABLE 7-1: Possible Divisions of 3 Items

YOU	FRIEND
A, B, C	—
A, B	C
A, C	B
B, C	A
A	B, C
B	A, C
C	A, B
—	A, B, C

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Notice that for every possible division there is another division with the items swapped between you and your friend. For example, in one division you get items A and B, and your friend gets item C. In the swapped division, your friend gets items A and B, and you get item C. Both of the matching divisions are equally even, so you can cut the number of possibilities you need to consider in half if you ignore one of each pair of divisions. One way to do that is to arbitrarily assign item A to you. Those sorts of tricks are what make algorithms fun!

That algorithm works well for small problems, but if the number of treasures is large, the algorithm will take too long. If there are N items, then there are 2^N possible ways to divide the treasure. (2^{N-1} possible ways if you arbitrarily give yourself the first item.)

For large values of N, the value 2^N can be large, for example, if you find a big treasure with 50 items, $2^N \approx 1.1 \times 10^{15}$. If you had a computer that could examine 1 million possible treasure divisions per second, it would take you almost 36 years to examine all the possibilities.

If you do find a larger treasure, you can't use the simple "try every possible solution" approach. In that case, you need to try a different algorithm.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In fact, this is known to be a provably difficult problem, and there are no known algorithms that can solve it exactly for large problem sizes. For large N, you need to turn to ***heuristics***—algorithms that give good solutions but that don't guarantee to give you the best solution possible.

For example, one heuristic would be to assign items randomly to you and your friend. The odds of you randomly guessing a perfect solution would be very small, but this method would be so fast you could perform several million or possibly even a billion trials and pick the best result you stumble across. (I think this is how countries set their economic policies. They make a bunch of random changes and, if any of them seem to work, they claim that was the plan all along.)

Another heuristic would be to give the next item to whichever of you currently has the smaller total value. For 50 items, that would require only 50 steps so it would be incredibly fast. The odds of you blundering across a perfect solution would still be fairly small, although the result would often be better than purely random guessing.

As this example shows, you need to understand how an algorithm will perform for your problem before you decide to use it. ***Big O notation*** is a system for studying the limiting behavior of algorithms as the size of the problem grows large. Search the Internet for "big O notation" or read an algorithms book (like the one I mentioned earlier) for more information on big O notation and algorithm complexity.

> **NOTE** *I can think of three other ways to divide the treasure perfectly evenly, and they don't even require a computer. First, donate the treasure to a museum. Second, auction off the treasure and split the proceeds. Finally, give it all to me and let me worry about it!*

Predictable

Some algorithms produce nice, predictable results every time they run. For example, if you search a list of numbers, you can find the largest one every time.

Other algorithms may be less predictable. The heuristics described in the previous section can't always find a perfect division of treasure. In fact, a perfect division may be impossible. (Suppose you have four treasures with values 10, 10, 20, and 30.) For the booty division problem, you can't even tell whether a perfect division of the spoils is *possible* without finding one.

Some algorithms may not produce the same results every time you run them. If you use the random heuristic described in the previous section several times, you'll probably get different answers each time. In that case, it may be hard to tell if the algorithm is working correctly.

It's also nice to know that an algorithm eventually finishes. It's a lot easier to tell that something's wrong when an algorithm takes twice as long to finish as you expect. Algorithms such as the random guessing heuristic can run indefinitely if you let them. In cases like that, you need to simply build in a cutoff that stops the algorithm after some set amount of time and takes the best solution found so far.

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

STOPPING CRITERIA

Actually, there are several ways you can stop an algorithm that might otherwise run indefinitely. For example, you might run until you find a solution of a particular quality. For the booty division problem, you might run until the algorithm finds a solution in which the two piles of treasure have values differing by no more than 10 percent. Of course, you'd still need to stop searching after some time period, just in case you can't find a solution that meets that criterion.

You can also save the best solution found after some time period and let the algorithm continue running in the background to look for better solutions. The application has a solution at all times, but it may gradually improve over time.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Although some algorithms such as this heuristic are inherently unpredictable, you should favor predictable algorithms if possible. It's much easier to debug a broken algorithm if you can reliably reproduce incorrect results whenever you need them.

Simple

Ideally an algorithm, like any other piece of code, should be elegantly simple. Simple code is easy to understand and easy to debug. It's easier to modify (if you decide to peel off the "Modify at your own risk" sticker) and it's easier to understand how the algorithm's performance varies for different inputs.

Some remarkably clever algorithms are also extremely simple, whereas others are a lot more involved. If you have a choice between a simple algorithm and a complex one that does the same job, pick the simple one.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Prepackaged

If you can find an algorithm that is implemented inside your programming language or in a library, use it. There's no need to write, test, debug, and maintain your own code if someone else can do it for you.

Prepackaged algorithms also tend to be more thoroughly studied and tested than anything you have time to write. A software vendor may spend hundreds of person-hours testing code that you would probably write, test, and shove out the door in a few hours. Its results may not always be better than yours, but if there is a problem you can ask the vendor to fix it instead of spending more time on it yourself.

Sometimes, libraries can also give you better performance. A library vendor may write more highly optimized code than you can. For example, its sorting routine might be written in assembly language whereas your version would be written in a higher-level language such as C++, C#, or Java.

In the end it may turn out that a prepackaged solution won't work for you either because it doesn't have the features you need or because your specific problem allows you to greatly improve the performance. However, it's always worth looking for an easier solution.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

TOP-DOWN DESIGN

If you can't find an algorithm to handle your situation, you need to write some code of your own. Even if you do find an algorithm that can be useful, you'll probably need to write some code to prepare for the algorithm and to process the results. So how do you get from a big, intimidating task like "design optimal routes for 300 delivery vehicles" or "schedule the classes for 1,200 middle-school students" to actual working code?

One useful approach is ***top-down design***, also called ***stepwise refinement***. In top-down design, you start with a high-level statement of a problem, and you break the problem down into more detailed pieces.

Next, you examine the pieces and break any that are too big into smaller pieces. You continue breaking pieces into smaller pieces until you have a detailed list of the steps you need to perform to solve the original problem.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

As you break a task into smaller pieces, you should be on the lookout for opportunities to save some work. If you notice that you're performing some chore more than once (perhaps while describing multiple main tasks), you should think about pulling that chore out and putting it in a separate method. Then all the tasks can use the same method. That not only lets you skip writing the same code a bunch of times, it also lets you invest extra time testing and debugging the common code while still saving time overall.

If the main task's description becomes too long, you should break it into shorter connected tasks. For example, suppose you need to write a method that searches a customer database for people who might be interested in golf equipment sales. You identify several dozen tests that identify likely prospects: people who earn more than $50,000 per year, people who live near golf courses, country club members, people who wear plaid shorts and sandals with spikes, and so forth.

If the list of tests is too long, it will be hard to read the full list of steps required to perform the original task. In that case, you should pull the tests out, place them in a new task described on a separate sheet of paper (or possibly several), and refer to the new task as a subtask of the original.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

For example, suppose the original method is called `PromoteSales`. Originally, its description might look like this:

`PromoteSale()`

1. Identify customers who are likely to buy items on sale and send them e-mails, flyers, or text messages as appropriate.

Now add some detail.

`PromoteSale()`

1. For each customer:
   A. If the customer is likely to buy:
      i. Send e-mail, flyer, or text message depending on the customer's preferences

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Step A "If the customer is likely to buy" will be pretty long, so create a new `IsCustomerLikelyToBuy` method. Similarly, step i will be fairly complicated, so create a new `SendSaleInfo` method. Now the main task looks like the following.

`PromoteSale()`

1. For each customer:
    A. If `IsCustomerIsLikelyToBuy()`
        i. `SendSaleInfo()`

At this point, you need to write the `IsCustomerLikelyToBuy` and `SendSaleInfo` methods. Here's the `IsCustomerLikelyToBuy` method.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

`IsCustomerLikelyToBuy()`

1. If (customer earns more than $50,000) return `true`.
2. If (customer lives within 1 mile of a golf course) return `true`.
3. If (customer is a country club member) return `true`.
4. If (customer wears plaid shorts and sandals with spikes) return `true`.

   ...

   73. If (none of the earlier was satisfied) return `false`.

Here's the `SendSaleInfo` method.

`SendSaleInfo()`

1. If (customer prefers e-mail) send e-mail message.
2. If (customer prefers snail-mail) send flyer.
3. If (customer prefers text messages) send text message.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

You can add other contact methods such as voicemail, telegraph, or carrier pigeon if appropriate.

This version of the `SendSaleInfo` method may also need some elaboration to explain how to determine which contact method the customer prefers.

`SendSaleInfo()`

1. Use the customer's `CustomerId` to look up the customer in the database's `Customers` table.
2. Get the customer's `PreferredContactMethod` value from the database record.
3. If (customer prefers e-mail) send e-mail message.
4. If (customer prefers snail-mail) send flyer.
5. If (customer prefers text messages) send text message.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Continue performing rounds of refinement, providing more detail for any steps that aren't painfully obvious, until the instructions are so detailed a fifth-grader could follow them.

At that point, sit down and write the code. If you've reached a sufficient level of detail, translating your instructions into code should be a mostly mechanical process.

INSUFFICIENT DETAIL

Some developers stop refining their code design when they think the list of instructions is enough to get them started but before it provides a painful level of detail. For example, many developers wouldn't bother to spell out how to look up the customer in the database and get the customer's `PreferredContactMethod` value.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

That's probably okay in this example, at least if you're an experienced developer. That kind of design shortcut can lead to problems, however, if a step turns out to be harder than you originally thought it would be.

It can be disastrous if you turn the instructions over to someone else who doesn't have your background and some steps are harder for that person than they would be for you. (I've worked on projects where the team lead gave a junior developer instructions that were obvious to the lead but mystifying to the developer. Rather than asking for help, the developer flailed about for weeks without making any progress.)

PROGRAMMING TIPS AND TRICKS

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Top-down design gives you a way to turn a task statement into code, but there are still a lot of tricks you can use to make writing code faster and easier. Other tips make it easier to test code, debug it when a problem surfaces, and maintain the code in the long term.

The following sections describe some of my favorite tips for writing good code.

Be Alert

Writing good code can be difficult. To know if you're writing the code correctly, you need to completely understand what you're trying to do, what the code actually does, and what could go wrong. You need to know in what situations the code might execute and how those situations could mess up your carefully laid plan. You need to ask yourself, what if an important file is locked, a needed value isn't found in a parameter table, or if a user can't remember his password.

Keeping everything straight can be quite a challenge. You can make your life a little easier if you write code only while you're wide awake and alert.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Most people have certain times of day when they're most alert. Some people are natural morning people and work best in the morning. Others work better in the afternoon. Some programmers do their best work after midnight when the rest of the world is asleep.

Figure out when your most effective hours are and plan to write code then. Fill out progress reports and timesheets during less productive hours.

Write for People, Not the Computer

Probably the most important tip in this chapter is to write code for people, not for computers. The computer doesn't care whether you use meaningful names for variables, indent your code nicely, use comments, or spell words correctly. It doesn't care how clever you are, and it doesn't care if your code produces a correct result.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In fact, the computer doesn't even read your code. Depending on your programming language and development environment, your code must be translated one, two, or more times before the computer can read it. All the computer wants to see is a string of 0s and 1s. If you were really writing code for the computer's benefit, your code would look like this:

```
10000000 00000000 00000000 00000000 00001110 00011111 10111010 00001110 00000000
10110100 00001001 11001101 00100001 10111000 00000001 01001100 11001101 00100001
01010100 01101000 01101001 01110011 00100000 01110000 01110010 01101111 01100111
01110010 01100001 01101101 00100000 01100011 01100001 01101110 01101110 01101111
01110100 00100000 01100010 01100101 00100000 01110010 01110101 01101110 00100000
01101001 01101110 00100000 01000100 01001111 01010011 00100000 01101101 . . .
```

The reason you write code in some higher-level programming language is that 0s and 1s are confusing for you. It would be incredibly difficult to remember the strings of 0s and 1s needed to represent different programming commands. (Although I know someone who used to have a computer's boot sequence memorized in binary so that he could toggle it in using switches when the computer needed to be restarted!)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Using a higher-level language lets you tell the computer what to do in a way that *you* can understand. Later, when your application is doing something wrong, it lets you trace through the execution to see what the computer is doing and why.

Debugging and maintaining code is far more difficult and time-consuming than writing code in the first place. The main reason is because you know what you are trying to do when you write code. Later when you're called upon to debug it, you might not remember exactly what the code is supposed to do. That makes it harder to identify the difference between what the code is supposed to do and what it actually does, so it's harder to fix.

Fixing a bug also has a much higher chance of adding a new bug than writing new code does, and for the same reason. When you're debugging, you don't have as clear an understanding of what the code is supposed to do. That makes it much easier to change the code in a way that breaks it.

To make debugging and maintaining code easier, you need to write code that is clear and easy to understand. Hopefully, whoever is eventually forced to track down a bug in your code won't be a violent psychopath, but you can make that person's job a lot easier if you remember it's that person you're writing for, not the computer.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

IT COULD BE YOU!

Always remember that the person debugging your code a year from now could be you! After enough time has passed, there's no way you'll remember exactly how the code was supposed to work. When you're writing the code initially, it may seem obvious, but a year or two later you'll only have whatever clues you left for yourself in the code to go by.

You can make your job easier by writing code that's clear and lucid, or you can learn to hate the younger you.

When you write code, remember that you're writing it for a possible future human reader (who might be you) and not for the computer.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Comment First

There are a few things that most programmers instinctively avoid because they don't feel like they're part of writing code. One of those is writing comments.

Many programmers write the bare minimum of comments they think they can get away with and then rush off to write more code. This is so common, in fact, that it has its own movement: just barely good enough (JBGE). The idea is that writing lots of comments is a waste of time. Besides, comments are usually wrong anyway, so rather than spending more time rewriting and fixing them, you should just write better code.

You can read my rant about JBGE in the section "Code Documentation" in Chapter 2. In this section, I want to talk about why comments need to be revised so often.

Many programmers use one of two models for writing comments. The first approach is to write comments as you code. You write a loop and then put a comment on top of it. Later you realize that the loop isn't quite right, so you change it and then update the comment. A bit later you realize that the loop still isn't right, so you change it again and revise the comment once more. After 37 rounds of revisions, you've either spent a huge amount of time updating the comment, or you've given up (thinking you'll revise the comment later) and the comment is hopelessly disconnected from the final code.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The second strategy is to write all the code without comments. When you're finished with your 37 revisions, you go back and insert the bare minimum number of comments that you think you can get away with without getting yelled at by the lead developer. (The lead developer does the same thing, so he doesn't care all that much about comments anyway.)

In both of these scenarios, the problem isn't that you have too many comments. The real problem is that you're trying to write comments to explain what the code *does* and not what it *should do*. When you tweak the code, you change what it does, so you need to update the comment. That creates a lot of work and that makes programmers reluctant to write comments.

If the code is well-written, the future reader will read the code to see what it actually does. What that person needs is comments to explain what the program is supposed to do. Then debugging becomes an exercise in determining where the program isn't doing what it's supposed to be doing.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

One way to write comments that explain what the program is supposed to be doing is to write the comments first. That lets you focus on the intent of the code and not get distracted by whatever code is sitting actually there in front of you.

It also means you don't need to revise the comment a dozen times. The code itself might change a dozen times, but the *intent* of the code better not! If it does, then you didn't do enough planning in the high-level and low-level design phases.

For example, consider the following C# code. (If you don't know C# or some similar language like C++ or Java, just focus on the comments.)

```
// Loop through the items in the "items" array.
for (int i = 0; i < items.Length - 1; i++)
{
    // Pick a random spot j in the array.
    int j = rand.Next(i, items.Length);
    // Save item i in a temporary variable.
    int temp = items[i];
    // Copy j into i.
    items[i] = items[j];
    // Copy temp into position k.
    items[j] = temp;
}
```

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The comments in this code explain what the code is doing, but they're mostly redundant. For example, the first comment explains exactly what the line of code that follows it does: through the array. That's certainly true, but any programmer who can't figure that out by looking at the looping statement itself probably shouldn't be debugging anyone's code.

Similarly, the other comments are just English versions of the programming statements that follow. The comment `Copy j into i` is even a bit cryptic, and the comment `Copy temp into position k` contains a typo, presumably because the code's author changed the name of a variable and forgot to update the comment.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

From a stylistic point of view, the comments are also distracting. They break up the visual flow and make the code look cluttered and busy.

Now that you've read the code, ask yourself, "What does it do?" Well yeah, it loops through the array, moves values into a temporary variable, and then moves them back into the array, but why? Does it accomplish what it was supposed to do? It's kind of hard to tell because the comments don't actually tell you what the code is supposed to do.

Now consider the following version of the code:

```
// Randomize the array.
// For each spot in the array, pick a random item and swap it into that spot.
for (int i = 0; i < items.Length - 1; i++)
{
    int j = rand.Next(i, items.Length);
    int temp = items[i];
    items[i] = items[j];
    items[j] = temp;
}
```

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

In this version, the comments tell you what the code is supposed to do, not what it actually does. The first comment gives the code's goal. The second comment tells how the code does it.

After you read the comments, you can read the code to see if it does what it's supposed to do. If you think there's a bug, you can step through the code in the debugger to see if it works as advertised.

This code is less cluttered and easier to read. It doesn't contain redundant comments that are just English versions of the code statements. These comments also don't need to be revised if the developer had to modify the code while writing it.

The best part of the comment-first approach is that the comments pop out for free if you use top-down code design. In the top-down method, you repeatedly break pieces of code into smaller and smaller pieces until you reach the point where a trained monkey could implement the code.

At that point, put whatever comment characters are appropriate for your language in front of the steps you've created (// for C#, C++, or Java; ' for Visual Basic; * for COBOL, and so forth), and drop them into the source code. Now fill in the code between the comments.

If your top-down design goes to a level of extreme detail, you may need to pull back a bit on the level of commenting. There's nothing wrong with the design going all the way to the level of explicitly giving the `if-then` statements you need to execute to perform a particular test, but that level of detail isn't necessary in the comments. Only include the comments that tell what the code is supposed to do and not the ones that repeat the actual code.

You may also need to add a few summary comments, particularly if your development team has rules for things like standard class and method headers, but most of the commenting work should be done.

You may also need to add a few comments to code that is particularly obscure and confusing. Remember, you might be debugging this code in a year or two.

Write Self-Documenting Code

In addition to writing good comments, you can make the code easier to read if you make the code self-documenting. Use descriptive names for classes, methods, properties, variables, and anything else you possibly can.

One exception to this rule is looping variables. Programmers often loop through a set of values and they use looping variables with catchy names like `i` or `j`. That's such a common practice that any programmer should be able to figure out what the variable means even though it doesn't have a descriptive name.

That doesn't mean you should avoid descriptive names if they make sense. If you're looping through the rows and columns of a matrix, you can name the looping variables `row` and `column`. Similarly, if you're looping through the pixels in an image, you can name the looping variables `x` and `y`. Those names give the reader just a little more information and make it easier to keep track of what the code is doing.

You can also make your code easier to understand if you don't use magic numbers. (A *magic number* is a value that just appears in the code with no explanation. For example, it might represent

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

an error code or database connection status.) Instead of using a magic number, use a named constant that has the same value.

Better still, if your language supports enumerated types, use them. They also give names to magic numbers and some development environments can use them to enforce type rules. For example, suppose you create an enumerated type named `MealSizes` that defines the values `Large`, `ExtraLarge`, and `Colossal`. Internally, the program might represent those values as 0, 1, and 2, but your code can use the textual values. If you define a variable `selected_size`, then your code can't give it the value 4 because that isn't an allowed value. (Actually, in many programs you can weasel around that check and force the variable to have the value 4. That would defeat the purpose of the enumerated type, so don't do it!)

Keep It Small

Write code in small pieces. Long pieces of code are harder to read. They require you to keep more information in your head at one time. They also require you to remember what was going on at the beginning of the code when you're reading statements much later.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

For example, suppose a piece of code loops through a set of customers. For each customer, it loops through the customer's orders. For each order, it loops through the order's items. Finally, for each item it loops through price points for that item. At some point later in the code, you'll come to statements that end each of those loops. For example, in C#, C++, or Java you'll come to a } character. If the code is short, you can look up a few lines to figure out which loop is ending. If the loops started a few hundred lines earlier, it may be hard to decide which loop is ending.

You may also eventually come across code like the following.

```
                }
            }
        }
    }
```

There's nothing here to tell you which loops are ending.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

THIS IS THE END

If a closing brace } is far from its corresponding opening brace {, you can make the code easier to understand by adding a comment after it explaining which loop is ending. For example, the following statement shows how you might end a `for` loop that's looping through the X coordinates of an image.

```
} // Next x
```

If you prefer more laconic comments, you could simply use `// x`.

I know some programmers loathe this style of comment, but if the start and end of a loop are far apart, this can be helpful.

I think many of the programmers who hate this kind of comment do so because they are forced to use it for *every* closing brace. You should use it only when it helps, not make an annoying rule that drives programmers crazy.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

If a piece of code becomes too long, break it into smaller pieces. Exactly how long is "too long" varies depending on what you're doing. Many developers used to break up methods that didn't fit on a one-page printout. A more recent tree-friendly rule of thumb is to break up a method if it won't fit on your computer's screen all at one time. (This may be why no one programs on smartphones. You'd have thousands of 10-line methods.)

AVOIDING BREAKUPS

Some complicated algorithms may be confusing enough that it's hard to keep everything they do in mind all at once, but splitting them can ruin performance. Or there may be no good place to split them because all the pieces are interrelated. In those cases, you may be stuck with a long chunk of code.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Sometimes, a little extra documentation can act as a roadmap to help you keep track of what the code is doing. (This should be documentation in a separate file, not just more comments, which would make the code even longer.)

You can also refer to external documentation inside the comments. For example, if your code uses Newton's method for finding the roots of a polynomial, don't embed a five-page essay in the comments. Instead add the following comment to the code and move on to something more productive.

```
// Use Newton's method to find the equation's roots. See:
// http://en.wikipedia.org/wiki/Newton's_method
```

In general, if it's hard to keep everything a method does in mind all at once, consider splitting it apart.

Stay Focused

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Each class should represent a single concept that's intuitively easy to understand. If you can't describe a class in a single sentence, then it's probably trying to do too much, and you should consider splitting it into several related classes.

For example, suppose you're writing an application to schedule seminars for a conference and to let people sign up for them. You probably shouldn't have a single class to represent attendees and presenters. Attendees and presenters may have a lot in common (they both have names, addresses, phone numbers, and e-mail addresses), but conceptually they are very different. Instead of creating a single `AttendeeOrPresenter` class to represent both kinds of person, make separate `Attendee` and `Presenter` classes. You can make them inherit from a common `Person` parent class, so you don't have to write the same name and address code twice, but making one mega-class will only confuse other developers. (Besides, the name `AttendeeOrPresenter` sounds wishy-washy.)

Just as a class should represent a single intuitive concept, a method should have a single clear purpose. Don't write methods that perform multiple unrelated tasks. Don't write a method called `PrintSalesReportAndFetchStockPrices`. The name might be nicely descriptive, but it's also cumbersome, so it's a hint that the method might not have a single clear purpose.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

One of my favorite examples of this was the `Line` method in earlier versions of Visual Basic. As you can probably guess, that method drew a line on a form or picture box. What's not obvious from the name is that it could also draw a box if you added the parameter `B` to the method call. I'm sure there was some implementation reason why this method drew boxes as well as lines, but seriously? A method named `Line` should draw lines not boxes.

Even if two tasks are related, it's often better to put them in separate methods so that you can invoke them separately if necessary.

Avoid Side Effects

A *side effect* is an unexpected result of a method call. For example, suppose you write a `ValidateLogin` method that checks a username and password in the database to see if the combination is valid. Oh, and by the way, it also leaves the application connected to the database. Leaving the database open is a side effect that isn't obvious from the name of the `ValidateLogin` method.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Side effects prevent a programmer from completely understanding what the application is doing. Because understanding the code is critical to producing high-quality results, avoid writing methods with side effects.

Sometimes, a method may need to perform some action that is secondary to its main purpose, such as opening the database before checking a username/password pair. There are several ways you can remove the hidden side effects.

First, you can make the side effect explicit in the method's name. For example, you could call this method `OpenDatabaseAndLogin`. That's not an ideal solution because the method isn't performing one well-focused task, but it's better than having unexpected side effects. (Any time you have "And" or "Or" in a method name, you may be trying to make the method do too much.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Second, the `ValidateLogin` method could close the database before it returns. That removes the hidden side effect; although it may reduce performance because you may want the database to be open for use by other methods.

Third, you could move the database opening code into a new method called `OpenDatabase`. The program would need to call `OpenDatabase` separately before it called `ValidateLogin`, but the process would be easy to understand.

Fourth, you could create an `OpenDatabase` method as before and make that method keep track of whether the database was already open. If the database is open, the method wouldn't open it again. Then you could make every method that needs the database (including `ValidateLogin`) call `OpenDatabase`. Methods such as `ValidateLogin` would encapsulate the call to `OpenDatabase` so you wouldn't need to think about it when you called `ValidateLogin`. There's still some extra work going on behind the scenes that you may not know about, but with this approach you don't need to keep track of whether the database is open or closed.

It may take a little extra work to remove side effects from a method, but it's worth it to make the code that calls the method easier to understand.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Validate Results

Murphy's law states, "Anything that can go wrong will go wrong." By that logic, you should always assume that your calculations will fail. Maybe not every single time, but sooner or later they will produce incorrect results.

Sometimes, the input data will be wrong. It may be missing or come in an incorrect format. Other times your calculations will be flawed. Values may not be correctly calculated or the results may be formatted incorrectly.

To catch these problems as soon as possible, you should add validation code to your methods. The validation code should look for trouble all over the place. It should examine the input data to make sure it's correct, and it should verify that the result your code produces is right. It can even verify that calculations are proceeding correctly in the middle of the calculation.

The main tool for validating code is the assertion. An *assertion* is a statement about the program and its data that is supposed to be true. If it isn't, the assertion throws an exception to tell you that something is wrong.

EXCEPTIONAL TERMINOLOGY

The term *exception* is programmer-speak for an unexpected error caused by the code. Exceptions can be caused by all sorts of situations such as trying to open a file that doesn't exist, trying to open a file that is locked by another program, performing an arithmetic calculation that divides by zero, using up all the computer's memory, or trying to use an object that doesn't exist.

When an exception occurs, the program's execution is interrupted. If you have an error handler in place, it can examine the exception information to figure out what went wrong and it can try to fix things. For example, it might tell the user to close the application that has a file locked and then it could try to open the file again.

If no error handler is ready to catch the exception, the program crashes.

For example, suppose you're writing a method to list customer orders sorted by their total cost. When the method starts, you could assert that the list contains at least two orders. You could also loop through the list and assert that every order has a total cost greater than zero.

After you sort the list, you could loop through the orders to verify that the cost of each order is at least as large as the cost of the one before it.

One type of assertion that can sometimes be useful is an invariant. An *invariant* is a state of the program and its data that should remain unchanged over some period of time.

For example, suppose you're working on a work scheduling application that defines an `Employee` class. You might decide that all `Employee` objects must always have at least 40 hours of work in any given week. (Although some of those hours might be coded as vacation.)

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Here the invariant condition is that the `Employee` object must have at least 40 hours of worked assigned to it. You could add assertions to the object's properties and methods to periodically verify that the invariant is still true. (Ideally, the class would provide only a few public properties and methods that could change the `Employee`'s work schedule and those would verify the invariant, at least before and after they do their work.)

TIMELY ASSERTIONS

Most programming languages have a method for conditional compilation. By setting a variable or flipping a switch, you can indicate that certain parts of the code shouldn't be compiled into the executable result. For example, the following code shows some validation code in C#.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

```
#if DEBUG_1
    // Validate the sorted order data.
    ...
#endif
```

The code between the `#if` and `#endif` directives is compiled only if the debugging symbol `DEBUG_1` is defined. If that symbol isn't defined, then the validation code is ignored by the compiler.

You can use techniques such as this one to add tons of validation code to the application. While you are testing and debugging the application, you can define the symbol `DEBUG_1` (and any other debugging symbols) so the testing code is compiled. When you're ready to release the program, you can remove the debugging symbols so that the program runs faster for the customers.

Later, if you discover a bug, you can redefine the debugging symbols to restore the testing code to hunt for the bug.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Some languages such as C# also have built-in conditional compilation for assertions. For example, the following statement asserts that an order's `TotalCost` value is greater than 0.

```
Debug.Assert(order.TotalCost > 0);
```

The compiler automatically includes this statement in debug builds and removes it from release builds.

Assertions and other validation code can make it easy to find bugs right after they are written when they're easest to fix. Unfortunately, it's hard to believe the code you just wrote isn't perfect. After all, you just spent hours slaving over a hot keyboard, pounding away with no breaks (maybe just one to refresh your coffee). The code is still fresh in your mind, so you know exactly how it works (or at least how you *think* it works). Obviously, there isn't bug in it or you would have already fixed it!

That thinking makes it hard for most programmers to write validation code. They just assume it isn't necessary.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

However, bugs do occur, so obviously they must be lurking in some of the code that was just written. If only you could convince programmers to add validation code to their methods, you might catch the bugs before they become established.

One way to encourage programmers to write validation code is to have them write it before writing the rest of a method's code. (This is similar to the way you can often get better comments if you write them before you write the code.) Writing the validation code first ensures that it happens.

This also has the advantage that you probably don't yet know exactly how the final code will work. You don't have it all in your head whispering seductively, "You did a great job writing me. There's really no need to validate the results." You also don't have preconceptions about how the code works, so you won't be influenced in how you write the validation code. You can look for incorrect results without making assumptions about where errors are impossible.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Practice Offensive Programming

The idea behind *defensive programming* is to make code work no matter what kind of garbage is passed into it for data. The code should work and produce some kind of result no matter what.

For example, consider the following `Factorial` function written in C#. (In case you don't remember, the factorial of a number N is written N! and equals $1 \times 2 \times 3 \times \ldots \times N$.)

```
public int Factorial(int number)
{
    int result = 1;
    for (int i = 2; i <= number; i++) result *= i;
    return result;
}
```

This code initializes the variable `result` to the value 1. It then multiplies that value by 2, 3, 4, and so on up to the number passed into the method as a parameter. It then returns `result`.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

This code works well in most cases. The code even works for strange values of the input parameter `number`. For example, if `number` is 0 or 1, the method sets `result` to 1, the loop does nothing, and the method returns the value 1. That happens to be correct because by definition 0! = 1 and 1! = 1.

If the parameter `number` is negative, the code also sets `result` to 1, the loop does nothing, and the method returns 1.

In fact, due to a quirk in the way C# handles integer overflow, this method even returns a value if `number` is really large. If `number` is 100, the loop causes `result` to overflow. The program sets `result` equal to 0, ignores the overflow, and continues merrily crunching away. When it's finished, it returns the value 0.

This is traditional defensive programming in action. No matter what value you pass into the method, it continues running. It may not always return a meaningful result, but it doesn't crash either.

Unfortunately this approach also hides errors. If the program is trying to calculate 100!, it's probably doing something wrong. At a minimum, it probably doesn't want to get the value 0.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

A better approach is to make the `Factorial` method throw a temper tantrum if its input is invalid. That way you know something is wrong and you can fix it. I call this, *offensive programming*. If something offends the code, it makes a big deal out of it.

The following code shows an offensive version of the `Factorial` method:

```
public int Factorial(int number)
{
    Debug.Assert(number >= 0);
    checked
    {
        int result = 1;
        for (int i = 2; i <= number; i++) result *= i;
        return result;
    }
}
```

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

The code begins with an assertion that verifies that the input parameter is at least 0.

The method includes the rest of its code in a `checked` block. The `checked` keyword tells C# to not ignore integer overflow and throw an exception instead. That takes care of cases in which the input parameter is too big.

If the program passes the new version of the Factorial function an invalid parameter, you'll know about it right away so you can fix it.

Use Exceptions

When a method has a problem, there are a couple ways to tell the program that something's wrong. Two of the most common methods are throwing an exception and passing an error code back to the calling code.

For example, the `Factorial` method shown in the previous section throws an exception if there's an error. The call to `Debug.Assert` throws an exception if its condition is `false`. The `checked` block throws an exception if the calculations cause integer overflow.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

As mentioned earlier in this chapter, an exception interrupts the program's execution and forces the code to take action. If you don't have any error handling code in place, the program crashes. That means a lazy programmer can't ignore a possible exception. If a method such as `Factorial` might throw an exception, the code must be prepared to handle it somehow.

In contrast, suppose the `Factorial` method indicated an error by returning an error code. For example, when passed the number –300, it might return the value –1. The factorial of a number is never negative, so the value –1 would indicate there is a problem.

The trouble with this approach is the program could ignore the error code. In that case, the program might end up displaying the bogus value –1 to the user or using that value in some other calculation. The result will be gibberish that is at best unhelpful and at worst misleading and confusing.

In general it's better to throw an exception to indicate an error instead of returning an error code. That way the program can't ignore a potentially confusing situation.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Write Exception Handers First

Now that you're using assertions and exceptions to indicate errors, the code that calls your method needs to use exception handling to deal with those exceptions.

Unfortunately, error handlers are a bit like comments in the sense that many programmers find them boring and don't like to write them. They're also a bit like validation code because it's easy to assume that they're not necessary because you *know* the code works.

One way to create better error handlers is to follow the same strategy you can use when writing comments and validation code: Do it first. When you start writing a method, paste in all the comments that you got from top-down design, add code to validate the inputs and verify the outputs, and then wrap error handling code around the whole thing.

First, make the error handling code look for exceptions that you expect to happen occasionally and that you can do something about (like trying to open a locked file).

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Next, add code that looks for other expected exceptions about which you can't do anything except complain to the user. That code should restate any exceptions in terms the user can understand. For example, instead of telling the user, "Arithmetic operation resulted in an overflow," you can present a more meaningful message like, "All orders must include at least 1 item."

Don't Repeat Code

If you find that you're writing the same (or nearly the same) piece of code more than once, consider moving it into a separate method that you can call from multiple places. That obviously saves you the time needed to write the code more than once. More important, it lets you debug and maintain the code in a single place.

Later if you need to modify the code for some reason, you need to make the change only in one method. If the code were duplicated, you would need to update it in every place it occurred. If you forgot to update it in one place, the different copies of the code would be out of synch and that can lead to some extremely confusing bugs. (Yes, I speak from experience here.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Defer Optimization

One of my favorite rules of programming is:

> *First make it work. Then make it faster if necessary.*

Highly optimized code can be a lot of fun to write, but it can also be very confusing. That means it takes longer to write and test. It's also harder to read, so it's harder to debug and fix if there is a problem.

Meanwhile, even the least optimized code is usually fast enough to get the job done. If you're displaying a list of 10 choices to the user, it doesn't matter if it takes 10 or 12 milliseconds to display. The user is going to stare at the choices for 3 or 4 seconds anyway, so it's not worth spending a lot of extra programming effort to shave 0.05 percent off the total time.

To program as efficiently as possible, write code in the most straightforward way you can, even if it's not the fastest way you can imagine. After you get the code working, you can decide whether it is so slow that it requires optimization.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

OPTIMIZATION OVERLOAD

I've never worked on a project that failed because the code was too slow. I've worked on a couple projects that were initially too slow and we rewrote their performance bottlenecks to bring them up to an acceptable speed. It really wasn't all that hard.

In contrast, I've worked on a couple projects that failed because their design was too complicated. People spent so much time trying to optimize the design and come up with the most efficient approach possible that the code was too complicated to implement and debug.

I'll say it again: First make it work. Then make it faster if necessary.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

If you do discover that the program isn't running fast enough, take some time to determine where performance improvements will give you the most benefit.

Typically 80 percent of a program's time is spent in 10 percent of the code. (Or 90 percent is spent in 10 percent of the code, or something. The idea is, the program spends most of its time executing a small fraction of the code.) Time you spend optimizing the 80 percent that's already fast enough is time that would be better spent on the slow 20 percent. (Frankly, you'd be better off just wasting that time by talking around the water cooler eating donuts. Time you spend messing about inside the 80 percent of the code that's already working fine can only make that code more confusing and harder to debug and maintain over time.)

Before you start ripping the code apart, use a profiler to see exactly where the problem code is. Then attack only the problem and not the whole program. (So you don't mess up the rest of the code with friendly fire.)

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

PROFILERS PROFILED

In case you haven't used one, a ***profiler*** is a program that monitors the progress of a program while it runs to identify the parts that are slow, that use the most memory, or that otherwise might be bottlenecks. Different profilers work in different ways. For example, some add code (called *instrumentation*) to your program to record the number of times every method is called and the amount of time the program spends in each method.

Profilers are *very* handy for tracking performance problems. I worked on one program that was taking approximately 20 minutes to load its data when it started. The project manager refused to buy a profiler ("real programmers don't need them") and had a number of theories about where in the data processing algorithms the bottlenecks were.

I snuck off into my office and installed a profiler for a 30-day free trial. Within a few hours, I had discovered that the problem wasn't in the main algorithms at all. The problem was actually in some fairly trivial string-processing code. Basically, the

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

program was going back to a database hundreds of times to re-fetch values that it had already loaded. I built a simple table to keep track of the values that had already been fetched and cut the program's startup time from 20 minutes to under 4.

If I hadn't used the profiler, I would probably have wasted a week or two and only shaved a minute or so off of the startup time. (After the fact the project lead admitted that, okay, perhaps a profiler was a good idea after all.)

Before you start optimizing code, make sure it works properly. Then if you do find that performance is insufficient, carefully analyze the problem (using a profiler if you can) so that you don't waste time optimizing code that is already fast enough.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

SUMMARY

Most programmers love to program, but they can't do a good job without the proper tools. If you don't have the right hardware, software, and network support, writing good code is slow and frustrating. That leads to distraction and more bugs. Writing good code also requires debugging, testing, and profiling tools. Depending on the development environment, you may also need code formatting and refactoring tools.

Before you starting writing code, make sure you have the tools you need to do so effectively. If you don't write code, make sure those who do get the tools they need.

Even if you're using all the proper tools, writing good code isn't guaranteed. There are dozens or perhaps hundreds of tips and tricks you can use to make your code safer. This chapter describes a few of my favorites. By using those techniques, you can make your programs more reliable, easier to debug, and easier to modify in the future.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Unfortunately, even the best program can still contain bugs. In fact, it's common in software engineering to assume that every nontrivial program contains some bugs. The only questions are, "How many bugs?" and "How often will the bugs affect the users?"

Testing lets you find and fix as many bugs as possible. If you test a program effectively, you can eventually reduce the number and severity of the remaining bugs so that the program is still usable. (Just as if you squash enough cockroaches, the rest eventually learn to hide better.)

The next chapter explains software testing. It describes techniques you can use to find bugs and estimate the number of bugs that remain in an application.

EXERCISES

1. The greatest common divisor (GCD) of two integers is the largest integer that evenly divides them both. For example, the GCD of 84 and 36 is 12 because 12 is the largest integer that evenly divides both 84 and 36. You can learn more about the GCD and the Euclidean algorithm, which you can find at `en.wikipedia.org/wiki/Euclidean_algorithm`.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Knowing that background, what's wrong with the comments in the following code? Rewrite the comments so that they are more effective. (Don't worry about the code if you can't understand it. Just focus on the comments.) (Hint: It should take you only a few seconds to fix these comments. Don't make a career out of it.)

```
// Use Euclid's algorithm to calculate the GCD.
private long GCD(long a, long b)
{
    // Get the absolute value of a and b.
    a = Math.Abs(a);
    b = Math.Abs(b);

    // Repeat until we're done.
    for (; ; )
    {
        // Set remainder to the remainder of a / b.
        long remainder = a % b;
        // If remainder is 0, we're done. Return b.
        if (remainder == 0) return b;
        // Set a = b and b = remainder.
        a = b;
        b = remainder;
    };
}
```

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

2. Why might you end up with the bad comments shown in the previous code?

3. How could you add validation code to the method shown in Exercise 1? (If you don't know how to write the validation code in C#, just indicate where it should be and what it should do.)

4. How could you apply offensive programming to the modified code you wrote for Exercise 3?

5. Should you add error handling to the modified code you wrote for Exercise 4?

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

6. The following code shows one way to swap the values in two integers `a` and `b`. The ^ operator takes the "exclusive or" (XOR) of the two values. The comments to the right explain how this method works.

```
// Swap a and b.     Let A and B be the original values.
b = a ^ b;        // b = A ^ B
a = a ^ b;        // a = A ^ (A ^ B) = (A ^ A) ^ B = B
b = a ^ b;        // b = B ^ (A ^ B) = (B ^ B) ^ A = A
```

This is a clever piece of code. It lets you swap two values without needing to waste memory for a temporary variable. So why isn't it good code? Write an improved version.

7. Using top-down design, write the highest level of instructions that you would use to tell someone how to drive your car to the nearest supermarket. (Keep it at a very high level.) List any assumptions you make.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

▸ WHAT YOU LEARNED IN THIS CHAPTER

- Use the right tools:
  - Fast development hardware and a fast Internet connection
  - A good development environment and source code formatters (if necessary)
  - Source code control
  - Profilers and static analysis tools
  - Testing and refactoring tools
  - Training
- Select algorithms that are effective, efficient, predictable, simple and (if possible) prepackaged.
- Use top-down design to fill in code details.
- Programming tips:
  - Program when you're most alert.
  - Write code for people, not for the computer.
  - Write comments, validation code, and exception handlers before you start writing the actual code.
  - Use descriptive names, named constants, and enumerated types.
  - Break long methods into manageable pieces.
  - Make each class represent a single concept that's intuitively easy to understand.
  - Keep methods tightly focused on a single task and without side effects.
  - Program offensively to expose bugs as quickly as possible.
  - Signal problems with exceptions instead of error codes.
  - If you're writing the same piece of code for a second time, extract it into a method that you can call repeatedly.
  - Only optimize after you sure it's necessary. Then use a profiler to find the code that actually needs optimization.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

424f7c283cced86ffd1b25db1e0b8e1b
ebrary

Stephens, Rod. Beginning Software Engineering (1st Edition). Somerset, US: Wrox, 2015. ProQuest ebrary. Web. 17 November 2015.
Copyright © 2015. Wrox. All rights reserved.