6

High-Level Design

> 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

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.

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.

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 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.

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. (Like the Three Stooges trying to walk through a door at the same time.)

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 to 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 are described in the section "Client/Server" 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 Android phone platform. In that case, the fact that your hardware platform is Android phones should probably be in the requirements. (You may also 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 and definitely not the last.

The high-level design should at least sketch out the kinds of security that the project will need during development and after implementation. You should consider problems that are intentional attacks from outside, bugs, and random mistakes (like losing a laptop or a flash drive containing a backup) that might compromise data.

You should also at least briefly consider problems that could be caused by that discontented team member who's been passed up for promotion five times (although you've read Chapter 3, "The Team" so your team is presumably running like a happy, well-oiled machine). If you work for a large company, you may also face hostility from employees who are not part of your coding clan team. (On one project, someone from another part of the company tried to delete all of our source code.)

You can protect the project from many of these issues by backing up the project documentation, source code, data, and anything else you can think of.

In the high-level design, sketch out any weaknesses that you spot and start protecting the project's assets immediately. Then fill in more details in the low-level design.

Security in general, both physical and electronic, is a large topic, so I'll defer a more detailed discussion until Chapter 8, "Security Design."

Hardware

Back in the old days when programmers worked by candlelight on treadle-powered computers, hardware options were limited. You pretty much wrote programs for large mainframes, and your choice of operating system (and sometimes even programming language) was determined by your choice of hardware.

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. A mini-computer acts sort of as a mini-mainframe that can serve a handful of users.

Wearable devices include such gadgets as computers strapped to the wearer's wrist (sort of like a cell phone with a wrist strap and possibly extra keys and buttons), wristbands, bracelets, watches, eyeglasses, monocles, headsets, rings, clothing, artificially intelligent hearing aids, and smart dog collars.

Appliances include refrigerators, doorbells, fire alarms, smart bicycles, door locks, toasters, medical sensors, voice-controlled devices, smart thermostats, lighting systems, printers, cameras, farm equipment, televisions, and who knows how many other kinds of devices form the *Internet of Things (IoT)*. These devices may communicate with each other or with your application.

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)
- IoT appliances

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, the IoT, or by some other method.

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 to tell you Fifi ran afoul of a skunk, an emergency dog-washer rushes to the scene with lights flashing and siren blaring.

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 more doggy shampoo.

For this application, you would specify the kind of phones the drivers will use (such as Android or iOS), 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, augmented reality glasses) to view their assignments.

User Interface

During high-level design, you can sketch out the user interface, but only at a high level. Save the details for later. For example, you can indicate the main methods for navigating through the application, but don't specify any exact screen layouts.

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, tapping if the hardware has a touch screen, or using key combinations such as Alt+Tab.

In contrast, some 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 forms to handle them all.

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, scroll bars, or text boxes).

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

FOLLOW EXISTING PRACTICES

Most users have a lot of experience with other 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 and/or left. That's the way Windows applications have been handling menus for decades and users are familiar with how they work.

continues

(continued)

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. If you plan to use these sorts of standard interactions, say so in the high-level design.

In contrast, 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 that you can click to open new forms. In that case, users will need to learn how to use your new system. That will lead to at least some confusion, and it might create a lot of annoyance for the users.

Nonstandard behavior may also disqualify an app for inclusion in some online stores. For example, Google Play, Microsoft Store, and Apple's App Store all have basic requirements. I'm not saying that you can't be creative, but you might want to have a plan B ready in case your app fails the store's testability or usability requirements.

(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 outbreaks of Tourette syndrome.)

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

You shouldn't specify every label and text box for every form during high-level design. You can handle that during low-level design and implementation (and after you've read Chapter 7, "Low-Level Design."). 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'll waste time squabbling about which approach is better. They'll 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.

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. Then, 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.

To get both teams working quickly, the high-level design specified an ASCII text file format that the teams could use to load test data. Instead of calling each other's code, the teams could write data into test files and then read data from those files. 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 moving data to and from 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 a temporary interface allowed both teams to work independently.

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

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 called Fishspedition. The application needs to assign a captain, first mate, cook, and barnacle scraper 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 or three captains and no first mate.) 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.)

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-based 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 asks you a sequence of hopefully relevant 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.

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 common architectures that you might want to use.

Monolithic

In a *monolithic architecture*, a single program does everything. It displays the user interface, accesses data, processes customer orders, prints invoices, orders lunch, 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.)

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. Probably.)

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

OS Monoliths *Operating systems tend to be monolithic with a single (albeit large) set of programs managing local logins, desktop, file system, task management, user access control, program access, program priorities, etc. (It's a major source of contention whether certain other programs such as a browser should be allowed as part of the OS.)*

This has the advantage of improving performance but has the disadvantage of sometimes making one program too dependent on another unrelated application. For example, if the File Explorer in Windows 10 or 11 crashes, it can drag the taskbar, desktop, and Alt+Tab Task Switcher down with it. In one test, I had to reboot to get everything back.

This problem has generally gotten better over the years (in the old days, many programs could summon the Blue Screen of Death), but these programs are still more tightly coupled than they would be had they been built by completely separate teams.

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.

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 6.1 shows this situation schematically.

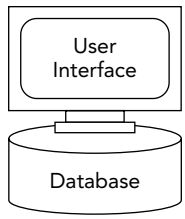

FIGURE 6.1: An application can directly hold its own data.

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 6.2 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.

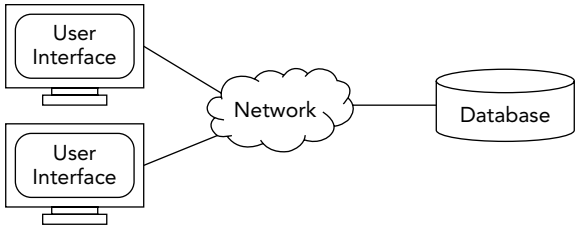

FIGURE 6.2: In a two-tier architecture, the client is separate from the server.

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.

The two-tier architecture makes it easier to support multiple clients with the same server, but it ties clients and servers 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.

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 6.3.

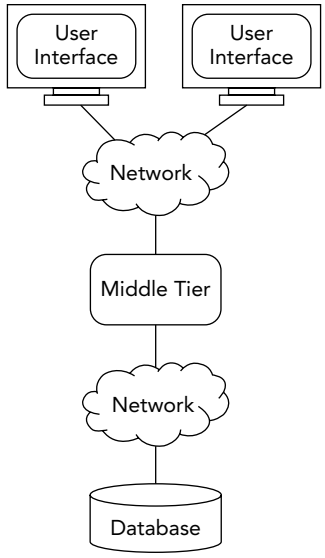

FIGURE 6.3: A three-tier architecture separates clients and servers with a middle tier.

In Figure 6.3, 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.

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. Similarly, if the client needs the data in a new format, the middle tier can translate the data to suit the client.

If the data format changes more drastically than just a formatting change (for example, if you need additional data that just isn't there), the middle tier can insert fake data until you have a chance to update the server to provide the actual data. That way development can continue on both the client and server while you straighten things out.

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.

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 server 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.

BEST PRACTICE

Multitier architectures are a best practice, largely because of the separation they provide between the client and server layers. Few applications 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.

An alternative would be to have the user interface ask components for that information, as shown in Figure 6.4. (Unified Modeling Language [UML] provides a more complex diagram for services that is described in the section "UML" later in this chapter.)

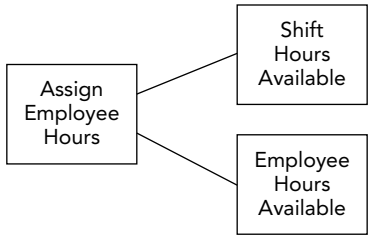

FIGURE 6.4: In a component-based architecture, components help decouple pieces of code.

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 also 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. Two common approaches for transmitting data across the web are SOAP and REST.

SOAP is a protocol that allows programs to invoke processes running on other computers, even computers that use a different operating system. In SOAP, requests are packaged in *Extensible Markup Language (XML)* where elements describe the function that the service should perform and any necessary parameters.

> ***A SOAPY FUTURE*** *SOAP was originally a backronym for Simple Object Access Protocol, but in version 1.2 that was scrapped and now it's just called SOAP. (Perhaps because they thought the S no longer applied and OAP would make a weird name?) Since then, many SOAP features have been deprecated and most development is moving in other directions, such as REST.*

> *In general, the term deprecate means to disapprove of something and discourage its use. In programming terms, it means that a vendor such as Microsoft still supports something but does not recommend using it and may discontinue support in the future. (Basically, they're saying, "Yes, we know that two years ago we told you that this was the wave of the future, but now that the future is here, we've changed our minds.")*

REST (representational state transfer) is an architectural style that was created to define how large-scale hypermedia systems such as the web should work. For example, a *RESTful* web service might send a request over the Internet via a specially formatted *URI (uniform resource locator)* and in response receive a *payload* containing data in XML, HTML, JSON, or some other data format.

In addition to the basic idea of a service, there are several variations with a service-oriented flavor.

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

Others define *microservices* as small services that you can glue together to build part or all of an application. As the name implies, the microservices are small, giving you benefits such as the ability to work on them separately, swap in new versions when necessary without disrupting the rest of the system, and use off-the-shelf microservices.

These ideas are also related to *aaS*, which stands for *as a service*. Broadly speaking, these include anything that is presented to the customer as a service. I suppose that includes pet-sitters and diaper services, but in this context it usually means some company wants to provide you some sort of software tool as a service. The term for talking about aaS when you don't know what service is being provided is *anything as a service (XaaS)*.

These services may be provided through methods such as libraries and programming interfaces, web-based control panels, or services in the SOAP and REST sense (in which case I suppose you could say that you had implemented "as a service" as a service).

There are many sub-genres of aaS that provide different types of service and that come with their own aaS acronyms. For example, artificial intelligence as a service (AIaaS), banking as a service (BaaS), database as a service (DBaaS), security as a service (SaaS), and so forth.

Two aaSs that are of particular interest to software development are infrastructure as a service and platform as a service.

Infrastructure as a service (IaaS) provides pieces of application infrastructure such as processors, network, storage, backups, security, recovery, firewalls, and so on.

Platform as a service (PaaS) provides everything that you need to develop software, such as compute power, compilers, tool libraries, and the ability to deploy applications to the cloud. This service may not include the ability to directly control the underlying cloud infrastructure, such as how the servers are arranged, where data is stored, or what operating system those servers use.

Microservices and XaaS are very popular with vendors eager to sell them to you. Search online for more details and to learn about more types of XaaS.

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 this 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 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.

ROTTEN RULES

For several years I had a fairly odd network connection leading directly to my phone company's central office, so one wire provided both phone and Internet service. 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 anymore.

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.

Rule-based systems are great for handling common simple scenarios, but when they encounter anything unexpected, they're 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. (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 to write and hard to debug. For example, suppose you're writing an application that sells office supplies such as staples, paper 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 6.1.

TABLE 6.1: 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 6.1, 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 copy of the balance to get $150, and then Purchaser 2 adds $10 to her copy of the balance to get $110.

Purchaser 1 then saves his new balance of $150 into the database. Finally, Purchaser 2 saves her 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 either way, you lose.)

A distributed architecture can improve performance as long as you don't run afoul of race conditions and a host of other confusing 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 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 5, "Requirements Gathering." (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.

- Monolithic—This is basically the default if none of the more elaborate architectures apply. We'll come back to this one later.
- 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.)

- 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.
- 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.
- Data-centric—The user defines the drawings, so there's no data around which to organize the program. (Although a more specialized program, perhaps an aerospace design program, might interact with data in a meaningful way.)
- 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. Or the user clicks to select a shape and then changes its properties or drags it to a new position.
- Rule-based—There are no rules that the user must follow to make a drawing, so this program isn't rule-based.
- 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 apply (such as multitier or service-oriented), this application can have a simple monolithic architecture with an event-driven user interface.

Reports

Almost any nontrivial software project can use some kind 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.

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 to stem the flood.

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 any 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 permanently 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.

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 (to the screen or to video files), output to special devices (such as electronic signs, thermostats, and other IoT devices), email, or text messages (which is as easy as sending an email to the right address). It could even send messages to pagers, if you can find any that aren't in museums.

> ***TIP*** *Text messages (or pager messages if you're old school) 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. I also knew a sys admin who had a program send her messages if her computer lab grew too hot.*

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, something more exotic such as a temporal database or object store, or a combination of those. Even a program that doesn't use databases 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.

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, Second Edition*, Wiley, 2023.) Although 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.

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 to 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.

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, views, 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.

Database Maintenance

A database is like a hall closet: Over time it gets disorganized and full of random junk like incorrect zip codes, out-of-service phone numbers, records of customers who moved out of state years ago, string, 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 or records that are no longer important. 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. You'll also need to design a way to retrieve the old data if you decide you want it back later.

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 (which you should also do for the cybersecurity reasons described in Chapter 8). In a low-priority application, that might involve copying a data file to a DVD or flash drive 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 (on multiple computers) every change made to any database record in real time. (One telephone company project I worked on even required the computers to be in widely separated locations so that they wouldn't all fail if a computer room was flooded or wiped out by an earthquake.)

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.

NoSQL

Relational databases are the workhorses of many applications, but there are other kinds of databases available that may be useful under some circumstances. NoSQL databases have become increasingly popular as computing power and data storage speed have improved.

Some people assume that *NoSQL* means "no SQL" (in other words, not a relational database), but others take it to mean "not only SQL." Either way, the interesting pieces are the things that NoSQL databases can do that relational databases do not. Typically a NoSQL database is assumed to be one of four types: a document database, a key-value store, a column-oriented database, or a graph database.

A *document database* is designed to store documents such as data stored in XML or JSON files. The data is relatively nonstructured so queries may have to search through much or all of the available data (hence the need for fast computers and speedy storage access).

A *key-value store* holds some sort of data that is fetchable by a key. For example, you might use a customer's name or ID to retrieve all of the information you have for that customer.

A *column-oriented database* (or *wide-column database*) stores data in columns much as a relational database does. The big difference is that a column-oriented database stores the columns separately. That makes it easy to distribute the data across many locations in the cloud. It also means you don't need to pull in all of the information for every row of data if you only need to study a few columns.

A *graph database* stores objects that are related to each other in interesting ways. For example, a social network map might contain nodes representing people with edges connecting people who know each other. You can store this sort of data in a relational database, but it's awkward and won't give you the same graph-oriented tools that a graph database can provide.

In general, NoSQL databases are flexible, work well with semi- and unstructured data, and can handle changes to requirements without the major reengineering that might be required with a relational database.

Of course, that lack of rigid structure is both a blessing and a curse. A relational database may contain more reliable data because it won't allow you to enter data that violates its rules and constraints.

If you're interested, look online for more information about NoSQL databases.

Cloud Databases

Cloud databases give you a new option for your database needs. Different cloud providers offer varying levels of backups, reliability, and expandability on demand, which can make scaling much easier. Some even offer data warehousing, audit trails, user access control, and more.

If you don't want to manage your databases personally, consider cloud data services. Make a list of your database needs and then shop around to see which vendors can meet those needs.

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, then 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 users can only modify the appropriate parameters. In many applications, only managers should change values such as the amounts and days that trigger late payment notices, but all users might be able to change their personal icon and font sizes or background colors.

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 email. Figure 6.5 shows one way you might diagram this data flow.

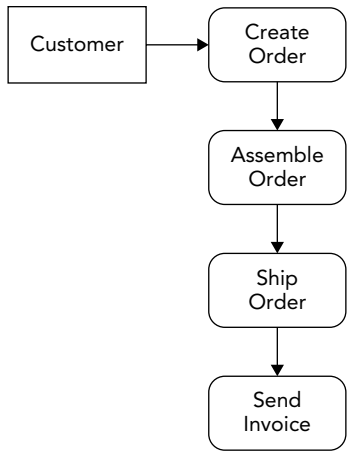

FIGURE 6.5: A data flow diagram shows how data such as a customer order flows through various processes.

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.

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 6.6 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

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.

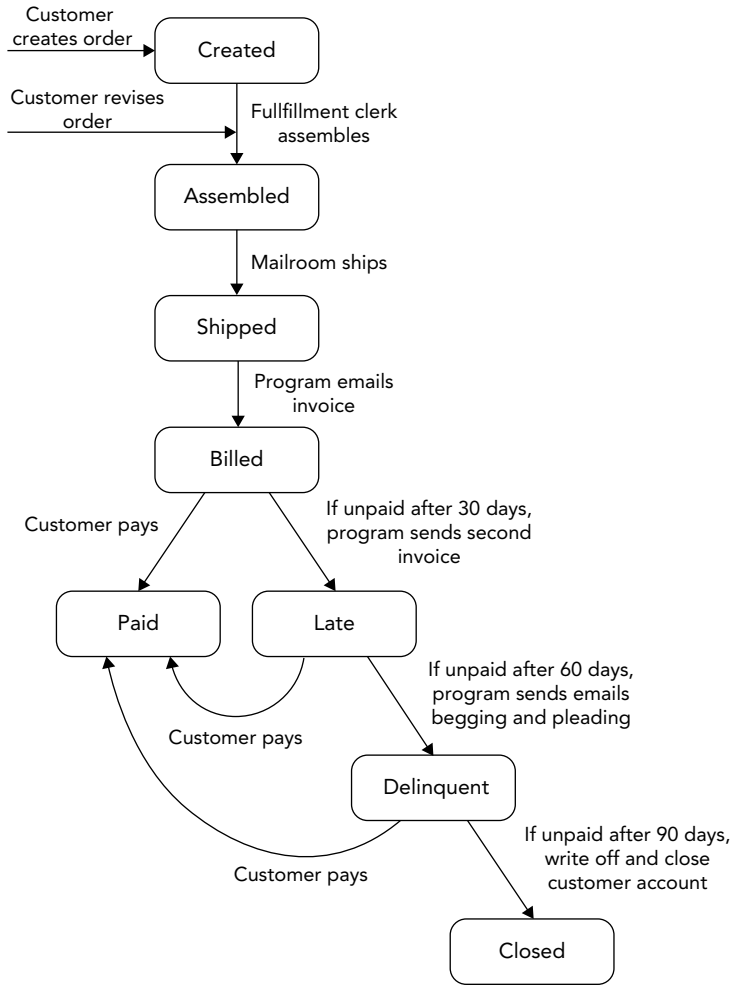

FIGURE 6.6: Complicated data flows may take different paths depending on circumstance.

Different people learn best in different ways, so you might want to provide several different kinds of training and either combine them or let the users pick those that are best for them. Supervised hands-on training is often effective because the users eventually need to use the application (that's why they're called users), and hands-on training is as close to actual use as possible.

During the high-level design, document the kinds of training that will be necessary. Trainers may be able to create content that discusses the application's high-level purpose now, but you'll have to fill in most of the details later as the project develops.

UML

As mentioned in Chapter 5, "Requirements 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`.)

UML 2.0 defines 14 diagram types divided into three categories as shown in the following list:

- Structure diagrams:
  - Class diagram
  - Component diagram
  - Composite structure diagram
  - Deployment diagram
  - Object diagram
  - Package diagram
  - Profile diagram
- Behavior diagrams:
  - Activity diagram
  - State machine diagram
  - Use case diagram
  - Interaction diagrams:
    - Communication diagram
    - Interaction overview diagram
    - Sequence 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. Search online for "UML" to get more information.

Structure Diagrams

A *structure diagram* describes things that will be in the system you are designing. For example, a class 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.
- 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.
- Deployment diagram—Describes the deployment of *artifacts* (files, scripts, executables, and the like) on *nodes* (hardware devices or execution environments that can execute artifacts).
- Object diagram—Focuses on a particular set of objects and their relationships at a specific time.
- 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.
- Profile diagram—Allows you to extend the UML syntax.

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 6.7 shows a simple diagram for the `Student` class.

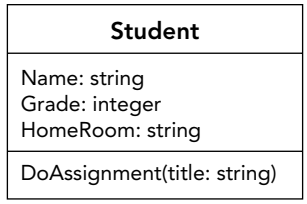

FIGURE 6.7: A class diagram describes the properties and methods of classes.

Some developers 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 6.7. You can also add the symbols shown in Table 6.2 to the left of a class member to show its visibility within the project.

TABLE 6.2: 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.

Class diagrams also often show relationships among classes with lines connecting related classes. A variety of line styles, symbols, arrowheads, and annotations gives 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 6.3 shows symbols you can add to the ends of a relationship.

TABLE 6.3: 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 6.8 shows the "is taking" relationship between the `Student` and `Course` classes. In that relationship, 1 `Student` object corresponds to 1 or more `Course` objects.

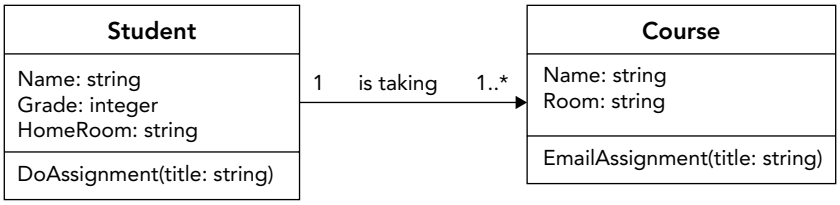

FIGURE 6.8: The relationship in this class diagram indicates that 1 student takes 1 or more courses.

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.

In a class diagram, you indicate inheritance by using a hollow arrowhead pointing from the child class to the parent class. Figure 6.9 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 one set of diagrams to show inheritance and another set of diagrams to show relationships.

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

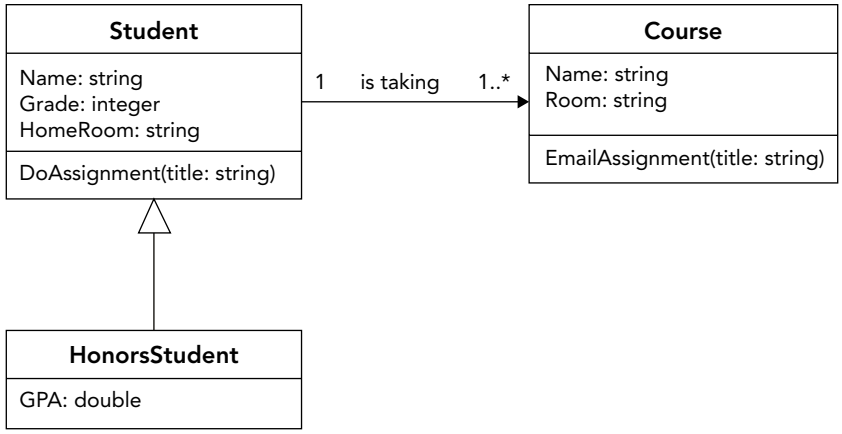

FIGURE 6.9: A class diagram indicates inheritance with a hollow arrowhead.

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.

Activity Diagrams

An *activity diagram* represents workflows for activities. They include several kinds of symbols connected with arrows to show the direction of the workflow. Table 6.4 summarizes the symbols.

TABLE 6.4: 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

Figure 6.10 shows a simple activity diagram for baking cookies.

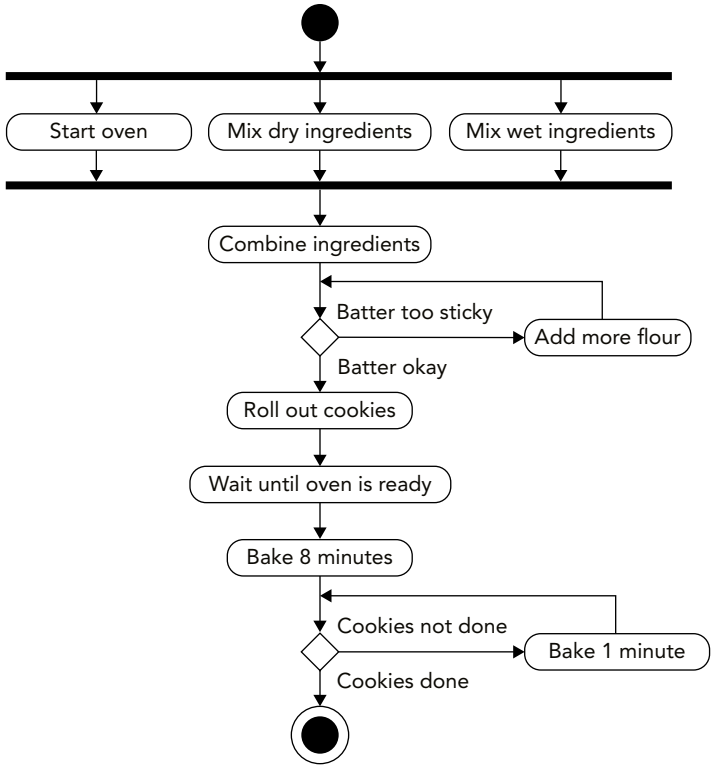

FIGURE 6.10: An activity diagram is a bit like a flowchart showing how work flows.

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 workflow 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.

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.

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 6.11 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.

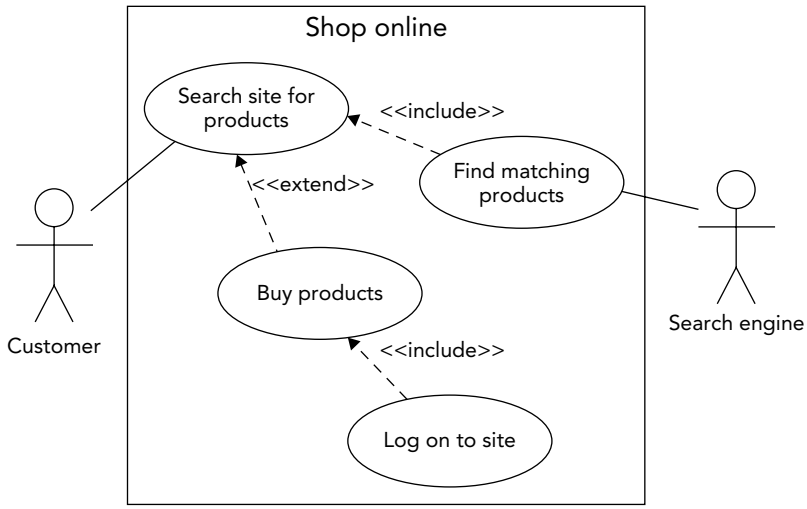

FIGURE 6.11: A use case diagram shows actors and the tasks they perform (possibly with subtasks and extensions).

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.

As in an activity diagram, a black circle represents the starting state and a circled black circle indicates the stopping state.

Figure 6.12 shows a simple state machine diagram for a program that reads a floating-point number (as in –17.32) followed by the Enter key.

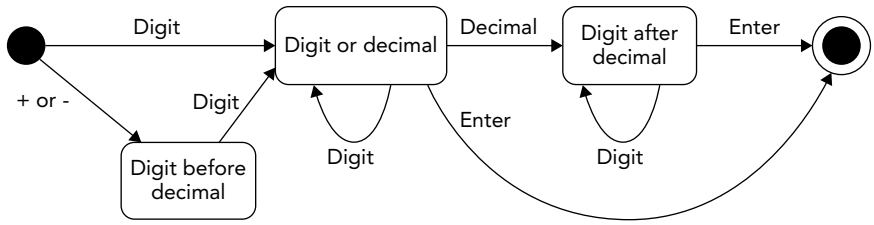

FIGURE 6.12: This state machine diagram represents reading a floating-point number.

When the program starts, it 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 playing an annoying sound and displaying an error message.) If it reads a + or –, the machine moves to the state "Digit before decimal."

From that state, the user must enter a digit, at which point the machine moves into the 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, the machine 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

Interaction diagrams form a subset of behavior 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.

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 6.13 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.

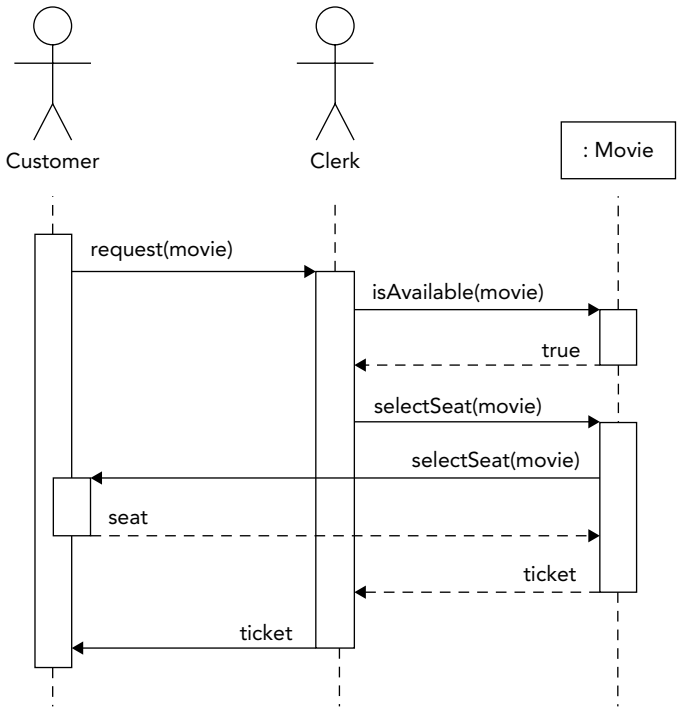

FIGURE 6.13: A sequence diagram shows the timing of messages between collaborating objects.

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.

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.

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 so you can follow the sequence.

Figure 6.14 shows a communication diagram for the movie ticket buying scenario that was shown in Figure 6.13.

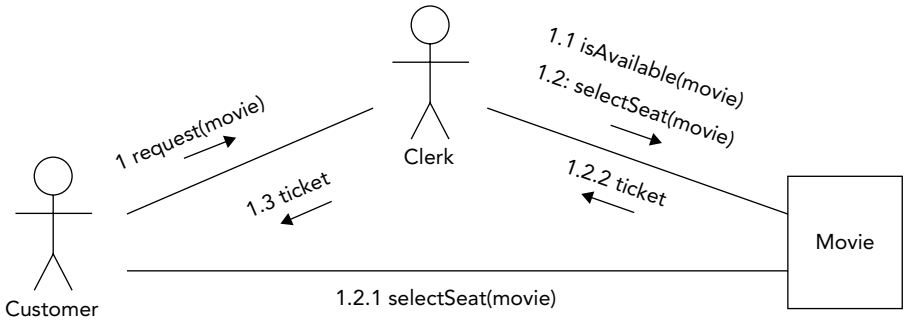

FIGURE 6.14: A communication diagram emphasizes the objects participating in a collaboration.

Following is the sequence of messages in Figure 6.14:

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 the way one or more objects change 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.

UML Summary

You may not need to use all of the UML diagrams for a particular application. For simple applications, you might not need any. Often well-written user stories, scripts, and use cases can better explain interactions to users who don't know UML.

Use whichever UML diagrams you find useful and describe other parts of the system in good old prose.

SUMMARY

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

- 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.

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 (noughts and crosses) against a simple computer opponent. It will display the user's high scores stored on the phone, not in an external database. Which architectures would be most appropriate and why?

3. Repeat Exercise 2 for a chess program running on a desktop, laptop, or tablet computer.
4. Repeat Exercise 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. Which of the following inputs would be accepted by the state machine diagram shown in Figure 6.12?
   - 1.23.45
   - 8
   - ++45
   - 1337
   - –1337
   - 1.2e34
   - (A blank string)
   - 67. (Ends in a decimal point)
9. 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 -12.3×10^{17}). Allow either E or e for the exponent symbol.

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.
- These are some of the things you should specify in a high-level design:
  - Important data that requires security
  - General security needs
  - Operating system (Windows, iOS, or Linux)
  - Hardware platform (desktop, laptop, tablet, phone, mainframe)
  - Other hardware (networks, printers, programmable signs, pagers, audio, video, IoT devices)
  - User interface style (navigational techniques, menus, screens versus forms)

- Internal interfaces
- External interfaces
- Architecture (monolithic, client-server, multitier, component-based, service-oriented, data-centric, event driven, rule-based, distributed)
- Reports (application usage, customer purchases, inventory, work schedules, productivity, ad hoc)
- Other outputs (printouts, web pages, data files, images, audio, video, email, text messages)
- Database (database platform, major tables and their relationships, auditing, user access, maintenance, backup, data warehousing)
- Top-level classes (`Customer`, `Employee`, and `Order`)
- Configuration data (algorithm parameters, due dates, expiration dates, durations, cutoff values)
- Data flows
- Training

- UML diagrams let 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.