CHAPTER 1

Software Engineering from 20,000 Feet

> *There are two ways of constructing a software design. One way is to make it so simple that there are obviously no deficiencies. The other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.*
>
> —C.A.R. HOARE

WHAT YOU WILL LEARN IN THIS CHAPTER:

- The basic steps required for successful software engineering
- Ways in which software engineering differs from other kinds of engineering
- How fixing one bug can lead to others
- Why it is important to detect mistakes as early as possible

In many ways, software engineering is a lot like other kinds of engineering. Whether you're building a bridge, an airplane, a nuclear power plant, or a new and improved version of Sudoku, you need to accomplish certain tasks. For example, you need to make a plan, follow that plan, heroically overcome unexpected obstacles, and hire a great band to play at the ribbon-cutting ceremony.

The following sections describe the steps you need to take to keep a software engineering project on track. These are more or less the same for any large project although there are some important differences. Later chapters in this book provide a lot more detail about these tasks.

REQUIREMENTS GATHERING

No big project can succeed without a plan. Sometimes a project doesn't follow the plan closely, but every big project must have a plan. The plan tells project members what they should be doing, when and how long they should be doing it, and most important what the project's goals are. They give the project direction.

One of the first steps in a software project is figuring out the requirements. You need to find out what the customers want and what the customers need. Depending on how well defined the user's needs are, this can be time-consuming.

WHO'S THE CUSTOMER?

Sometimes, it's easy to tell who the customer is. If you're writing software for another part of your own company, it may be obvious who the customers are. In that case, you can sit down with them and talk about what the software should do.

In other cases, you may have only a vague notion of who will use the finished software. For example, if you're creating a new online card game, it may be hard to identify the customers until after you start marketing the game.

Sometimes, you may even be the customer. I write software for myself all the time. This has a lot of advantages. For example, I know exactly what I want and I know more or less how hard it will be to provide different features. (Unfortunately, I also sometimes have a hard time saying "no" to myself, so projects can drag on for a lot longer than they should.)

In any project, you should try to identify your customers and interact with them as much as possible so that you can design the most useful application possible.

After you determine the customers' wants and needs (which are not always the same), you can turn them into requirements documents. Those documents tell the customers what they will be getting, and they tell the project members what they will be building.

Throughout the project, both customers and team members can refer to the requirements to see if the project is heading in the right direction. If someone suggests that the project should include a video tutorial, you can see if that was included in the requirements. If this is a new feature, you might allow that change if it would be useful and wouldn't mess up the rest of the schedule. If that request doesn't make sense, either because it wouldn't add value to the project or you can't do it with the time you have, then you may need to defer it for a later release.

CHANGE HAPPENS

Although there are some similarities between software and other kinds of engineering, the fact that software doesn't exist in any physical way means there are some major differences as well. Because software is so malleable, users frequently ask for new features up to the day before the release party. They ask developers to shorten schedules and request last-minute changes such as switching database platforms or even hardware platforms. (Yes, both of those have happened to me.) "The program is just 0s and 1s," they reason. "The 0s and 1s don't care whether they run on an Android tablet or a Windows Phone, do they?"

In contrast, a company wouldn't ask an architectural firm to move a new convention center across the street at the last minute; a city transportation authority wouldn't ask the builder to add an extra lane to a freeway bridge right after it opens; and no one would try to insert an atrium level at the bottom of a newly completed 90-story building.

HIGH-LEVEL DESIGN

After you know the project's requirements, you can start working on the high-level design. The high-level design includes such things as decisions about what platform to use (such as desktop, laptop, tablet, or phone), what data design to use (such as direct access, 2-tier, or 3-tier), and interfaces with other systems (such as external purchasing systems).

The high-level design should also include information about the project architecture at a relatively high level. You should break the project into the large chunks that handle the project's major areas of functionality. Depending on your approach, this may include a list of the modules that you need to build or a list of families of classes.

For example, suppose you're building a system to manage the results of ostrich races. You might decide the project needs the following major pieces:

- Database (to hold the data)
- Classes (for example, Race, Ostrich, and Jockey classes)
- User interfaces (to enter Ostrich and Jockey data, enter race results, produce result reports, and create new races)

- External interfaces (to send information and spam to participants and fans via e-mail, text message, voice mail, and anything else we can think of)

You should make sure that the high-level design covers every aspect of the requirements. It should specify what the pieces do and how they should interact, but it should include as few details as possible about how the pieces do their jobs.

TO DESIGN OR NOT TO DESIGN, THAT IS THE QUESTION

At this point, fans of extreme programming, Scrum, and other incremental development approaches may be rolling their eyes, snorting in derision and muttering about how those methodologies don't need high-level designs.

Let's defer this argument until Chapter 5, "High-Level Design," which talks about high-level design in greater detail. For now, I'll just claim that every design methodology needs design, even if it doesn't come in the form of a giant written design specification carved into a block of marble.

LOW-LEVEL DESIGN

After your high-level design breaks the project into pieces, you can assign those pieces to groups within the project so that they can work on low-level designs. The low-level design includes information about *how* that piece of the project should work. The design doesn't need to give every last nitpicky detail necessary to implement the project's major pieces, but they should give enough guidance to the developers who will implement those pieces.

For example, the ostrich racing application's database piece would include an initial design for the database. It should sketch out the tables that will hold the race, ostrich, and jockey information.

At this point you will also discover interactions between the different pieces of the project that may require changes here and there. The ostrich project's external interfaces might require a new table to hold e-mail, text messaging, and other information for fans.

DEVELOPMENT

After you've created the high- and low-level designs, it's time for the programmers to get to work. (Actually, the programmers should have been hard at work gathering requirements, creating the high-level designs, and refining them into low-level designs, but development is the part that most programmers enjoy the most.) The programmers continue refining the low-level designs until they know how to implement those designs in code.

(In fact, in one of my favorite development techniques, you basically just keep refining the design to give more and more detail until it would be easier to just write the code instead. Then you do exactly that.)

As the programmers write the code, they test it to make sure it doesn't contain any bugs.

At this point, any experienced developers should be snickering if not actually laughing out loud. It's a programming axiom that no nontrivial program is completely bug-free. So let me rephrase the previous paragraph.

As the programmers write the code, they test it to find and remove as many bugs as they reasonably can.

TESTING

Effectively testing your own code is extremely hard. If you just wrote the code, you obviously didn't insert bugs intentionally. If you knew there was a bug in the code, you would have fixed it before you wrote it. That idea often leads programmers to assume their code is correct (I guess they're just naturally optimistic) so they don't always test it as thoroughly as they should.

Even if a particular piece of code is thoroughly tested and contains no (or few) bugs, there's no guarantee that it will work properly with the other parts of the system.

One way to address both of these problems (developers don't test their own code well and the pieces may not work together) is to perform different kinds of tests. First developers test their own code. Then testers who didn't write the code test it. After a piece of code seems to work properly, it is integrated into the rest of the project, and the whole thing is tested to see if the new code broke anything.

Any time a test fails, the programmers dive back into the code to figure out what's going wrong and how to fix it. After any repairs, the code goes back into the queue for retesting.

A SWARM OF BUGS

At this point you may wonder why you need to retest the code. After all, you just fixed it, right?

Unfortunately fixing a bug often creates a new bug. Sometimes the bug fix is incorrect. Other times it breaks another piece of code that depended on the original buggy behavior. In the known bug hides an unknown bug.

Still other times the programmer might change some correct behavior to a different correct behavior without realizing that some other code depended on the original correct behavior. (Imagine if someone switched the arrangement of your hot and cold water faucets. Either arrangement would work just fine, but you may get a nasty surprise the next time you take a shower.)

Any time you change the code, whether by adding new code or fixing old code, you need to test it to make sure everything works as it should.

Unfortunately, you can never be certain that you've caught every bug. If you run your tests and don't find anything wrong, that doesn't mean there are no bugs, just that you haven't found them. As programming pioneer Edsger W. Dijkstra said, "Testing shows the presence, not the absence of bugs." (This issue can become philosophical. If a bug is undetected, is it still a bug?)

The best you can do is test and fix bugs until they occur at an acceptably low rate. If bugs don't bother users too frequently or too severely when they do occur, then you're ready to move on to deployment.

EXAMPLE **Counting Bugs**

Suppose requirements gathering, high-level design, low-level design, and development works like this: Every time you make a decision, the next task in the sequence includes two more decisions that depend on the first one. For example, when you make a requirements decision, the high-level design includes two decisions that depend on it. (This isn't exactly the way it works, but it's not as ridiculous as you might wish.)

Now suppose you made a mistake during requirements gathering. (The customer said the application had to support 30 users with a 5-second response time, but you heard 5 users with a 30-second response time.)

If you detect the error during the requirements gathering phase, you need to fix only that one error. But how many incorrect decisions could depend on that one mistake if you don't discover the problem until after development is complete?

The one mistake in requirements gathering leads to two decisions in high-level design that could be incorrect.

Each of the two possible mistakes in high-level design leads to two new decisions in low-level design that could also be wrong, giving a total of $2 \times 2 = 4$ possible mistakes in low-level design.

Each of the four suspicious low-level design decisions lead to two more decisions during development, giving a total of $4 \times 2 = 8$ possible mistakes during development.

Adding up all the mistakes in requirements gathering, high-level design, low-level design, and development gives a total of $1 + 2 + 4 + 8 = 15$ possible mistakes. Figure 1.1 shows how the potential mistakes propagate.

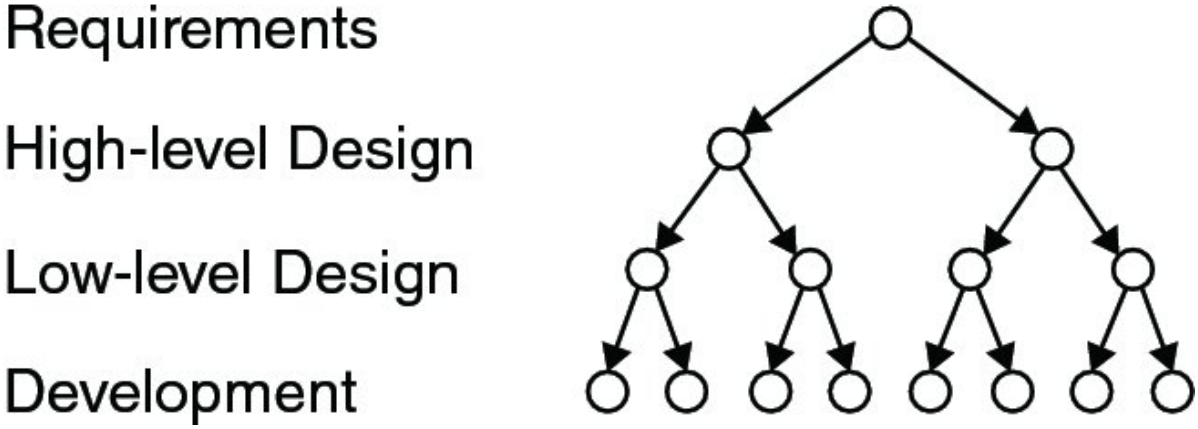

Figure 1.1 The circles represent possible mistakes at different stages of development. One early mistake can lead to lots of later mistakes.

In this example, you have 15 times as many decisions to track down, examine, and possibly fix than you would have if you had discovered the mistake right away during requirements gathering. That leads to one of the most important rules of software engineering. A rule that is so important, I'll repeat it later in the book:

> *The longer a bug remains undetected, the harder it is to fi x.*

Some people think of testing as something you do after the fact to verify that the code you wrote is correct. Actually, testing is critical at every stage of development to ensure the resulting application is usable.

DEPLOYMENT

Ideally, you roll out your software, the users are overjoyed, and everyone lives happily ever after. If you've built a new variant of Tetris and you release it on the Internet, your deployment may actually be that simple.

Often, however, things don't go so smoothly. Deployment can be difficult, time-consuming, and expensive. For example, suppose you've written a new billing system to track payments from your company's millions of customers. Deployment might involve any or all of the following:

- New computers for the back-end database
- A new network
- New computers for the users
- User training

- On-site support while the users get to know the new system
- Parallel operations while some users get to know the new system and other users keep using the old system
- Special data maintenance chores to keep the old and new databases synchronized
- Massive bug fixing when the 250 users discover dozens or hundreds of bugs that testing didn’ t uncover
- Other nonsense that no one could possibly predict

WHO COULD HAVE PREDICTED?

I worked on one project that assigned repair people to fix customer problems for a phone company. Twice during live testing the system assigned someone to work at his ex-wife’s house. Fortunately, the repair people involved recognized the address and asked their supervisors to override the assignments.

If psychics were more consistent, it would be worth adding one to every software project to anticipate these sorts of bizarre problems. Failing that or a working crystal ball, you should allow some extra time in the project schedule to handle these sorts of completely unexpected complications.

MAINTENANCE

As soon as the users start pounding away on your software, they’ll find bugs. (This is another software axiom. Bugs that were completely hidden from testers appear the instant users touch the application.)

Of course, when the users find bugs, you need to fix them. As mentioned earlier, fixing a bug sometimes leads to another bug, so now you get to fix that one as well.

If your application is successful, users will use it a lot, and they’ll be even more likely to find bugs. They also think up a slew of enhancements, improvements, and new features that they want added immediately.

This is the kind of problem every software developer wants to have: customers that like an application so much, they're clamoring for more. It's the goal of every software engineering project, but it does mean more work.

WRAP-UP

At this point in the process, you're probably ready for a break. You've put in long hours of planning, design, development, and testing. You've found bugs you didn't expect, and the users are keeping you busy with bug reports and change requests. You want nothing more than a nice, long vacation.

There's one more important thing you should do before you jet off to Cancún: You need to perform a post-mortem. You need to evaluate the project and decide what went right and what went wrong. You need to figure out how to make the things that went well occur more often in the future. Conversely, you need to determine how to prevent the things that went badly in the future.

Right after the project's completion, many developers don't feel like going through this exercise, but it's important to do right away before everyone forgets any lessons that you can learn from the project.

EVERYTHING ALL AT ONCE

Several famous people have said, "Time is nature's way to keep everything from happening all at once." Unfortunately, time doesn't work that way in software engineering. Depending on how big the project is and how the tasks are distributed, many of the basic tasks overlap—and sometimes in big ways.

Suppose you're building a huge application that's vital to national security interests. For example, suppose you want to optimize national energy drink ordering, distribution, and consumption. This is a big problem. (Really, it is.) You might have some ideas about how to start, but there are a lot of details that you'll need to work out to build the best possible solution. You'll probably need to spend quite a while studying existing operations to develop the user requirements.

You could spend several weeks peppering the customers with questions while the rest of the development team plays *Mario Cart* and consumes the drinks you're studying, but that would be inefficient.

A better use of everyone's time would be to put people to work with as much of the project that is ready to roll at any given moment. Several people can work with the customers to define the requirements. This takes more coordination than having a single person gather requirements, but on big projects it can still save you a lot of time.

After you think you understand some of the requirements, other team members can start working on high-level designs to satisfy them. They'll probably make more mistakes than they would if you waited until the requirements are finished, but you'll get things done sooner.

As the project progresses, the focus of work moves down through the basic project tasks. For example, as requirements gathering nears completion, you should finalize the high-level designs, so team members can move on to low-level designs and possibly even some development.

Meanwhile, throughout the entire project, testers can try to shoot holes in things. As parts of the application are finished, they can try different scenarios to make sure the application can handle them.

Depending on the testers' skills, they can even test things such as the designs and the requirements. Of course, they can't run the requirements through a compiler to see if the computer can make sense of them. They can, however, look for situations that aren't covered by the requirements. ("What if a shipment of Quickstart Energy Drink is delayed, but the customer is on a cruise ship and just crossed the International Date Line! Is the shipment still considered late?")

Sometimes tasks also flow backward. For example, problems during development may discover a problem with the design or even the requirements. The farther back a correction needs to flow, the greater its impact. Remember the earlier example where every problem caused two more? The requirements problem you discovered during development could lead to a whole slew of other undiscovered bugs. In the worst case, testing of "finished" code may reveal fundamental flaws in the early designs and even the requirements.

REQUIREMENT REPAIRS

The first project I worked on was an inventory system for NAVSPECWARGRU (Navy Special Warfare Group, basically the Navy SEALs). The application let you define equipment packages for various activities and then let team members check out whatever was necessary. (Sort of the way a Boy Scouts quartermaster does this. For this campout, you'll need a tent, bedroll, canteen, cooking gear, and M79 grenade launcher.)

Anyway, while I was building one of the screens, I realized that the requirements specifications and high-level design didn't include any method for team members to return equipment when they were done with it. In a matter of weeks, the quartermaster's warehouse would be empty and the barracks would be packed to the rafters with ghillie suits and snorkels!

This was a fairly small project, so it was easy to fix. I told the project manager, he whipped up a design for an inventory return screen, and I built it. That kind of quick correction isn't possible for every project, particularly not for large ones, but in this case the whole fix took approximately an hour.

In addition to overlapping and flowing backward, the basic tasks are also sometimes handled in very different ways. Some development models rely on a specification that's extremely detailed and rigid. Others use specifications that change so fluidly it's hard to know whether they use any specification at all. Iterative approaches even repeat the same basic tasks many times to build ever-improving versions of the final application. The chapters in the second part of this book discuss some of the most popular of those sorts of development approaches.

SUMMARY

All software engineering projects must handle the same basic tasks. Different development models may handle them in different ways, but they're all hidden in there somewhere.

In fact, the strengths and weaknesses of various development models depend in a large part on how they handle these tasks. For example, agile methods and test-driven development use

frequent builds to force developers to perform a lot of tests early on so that they can catch bugs as quickly as possible. (For a preview of why that's important, see the "Counting Bugs" example earlier in this chapter and Exercise 4.)

The chapters in Part II, "Development Models," describe some of the most common development models. Meanwhile the following chapters describe the basic software engineering tasks in greater detail. Before you delve into the complexities of requirements gathering, however, there are a few things you should consider.

The next chapter explains some basic tools that you should have in place before you consider a new project. The chapter after that discusses project management tools and techniques that can help you keep your project on track as you work through the basic software engineering tasks.

EXERCISES

1. What are the basic tasks that all software engineering projects must handle?
2. Give a one sentence description of each of the tasks you listed for Exercise 1.
3. I have a few customers who do their own programming, but who occasionally get stuck and need a few pointers or a quick example program. A typical project runs through the following stages:
    a. The customer sends me an e-mail describing the problem.
    b. I reply telling what I think the customer wants (and sometimes asking for clarification).
    c. The customer confirms my guesses or gives me more detail.
    d. I crank out a quick example program.
    e. I e-mail the example to the customer.
    f. The customer examines the example and asks more questions if necessary.
    g. I answer the new questions.

    Earlier in this chapter, I said that every project runs through the same basic tasks. Explain where those tasks are performed in this kind of interaction. (For example, which of those steps includes testing?)
4. List three ways fixing one bug can cause others.
5. List five tasks that might be part of deployment.

WHAT YOU LEARNED IN THIS CHAPTER

- All projects perform the same basic tasks:
    1. Requirements Gathering
    2. High-level Design
    3. Low-level Design
    4. Development
    5. Testing
    6. Deployment
    7. Maintenance
    8. Wrap-up
- Different development models handle the basic tasks in different ways, such as making some less formal or repeating tasks many times.
- The basic tasks often occur at the same time, with some developers working on one task while other developers work on other tasks.
- Work sometimes flows backward with later tasks requiring changes to earlier tasks.
- Fixing a bug can lead to other bugs.
- The longer a mistake remains undetected, the harder it is to fix.
- Surprises are inevitable, so you should allow some extra time to handle them.