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.

continues

(continued)

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 × 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 × 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.

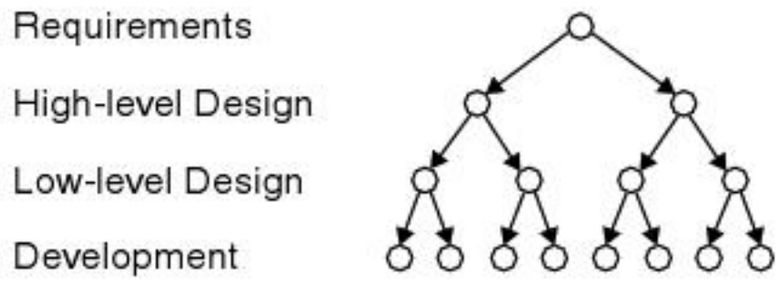

FIGURE 1-1: The circles represent possible mistakes at different stages of development. One early mistake can lead to lots of later mistakes.

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 fix.

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.

2

Before the Beginning

It's not whether you win or lose, it's how you place the blame.

—Oscar Wilde

WHAT YOU WILL LEARN IN THIS CHAPTER:

- The features that a document management system provides
- Why documentation is important
- How you can easily archive e-mails for later use
- Typical types of documentation

Before you start working on a software project, even before you dig into the details of what the project is about, there are preparations you should make. In fact, some of these can be useful even if you're not considering a software project.

These tools improve your chances for success in any complicated endeavor. They raise the odds that you'll produce something that will satisfy the application's customers. They'll also help you survive the process so that you'll still be working on the project when the accolades start rolling in.

Typically, you'll use these tools and techniques throughout all of a project's stages. You'll use them while you're gathering requirements from the customer, during the design and programming phases, and as you roll out the final result to the users. You'll even use them after you've finished releasing an application and you're considering enhancements for the next version.

The following sections describe some beginning-to-end tools that you can use to help keep team members focused and the project on track.

DOCUMENT MANAGEMENT

A software engineering project uses a lot of documents. It uses requirements documents, use cases, design documents, test plans, user training material, lunch menus for team-building exercises, resumes if the project doesn't go well, and much more. (I'll describe these kinds of documentation in later chapters.) Even a relatively modest project could have hundreds or even thousands of pages of documentation.

To make matters more confusing, many of those are "living" documents that evolve over time. In some projects, the requirements are allowed to change as the project progresses. As developers get a better sense for which tasks will be hard and which will be easy, the customers may want to revise the requirements to include new, simple features and eliminate old, complicated features.

As the project progresses, the customers will also get a better understanding of what the system will eventually do and they may want to make changes. They may see some partially implemented feature and decide that it isn't that useful. They may even come up with new features that they just plain forgot about at the beginning of the project. ("I know we didn't *explicitly* say you need a way to log into the system, but I'm quite sure that's going to be necessary at some point.")

CHANGE CONTROL

If you let everyone make changes to the requirements, how can you meet them? Just when you satisfy one requirement, someone can change it, so you're not done after all. (Imagine running after the bus in the rain while the driver cackles evilly and checks the side mirror to make sure he's going just a little faster than you're running.) Eventually, the requirements need to settle down so that you can achieve them.

Allowing everyone to change the requirements can also result in muddled, conflicting, and confusing goals and designs. This is more or less how laws and government spending bills are written, so it shouldn't be a surprise that the results aren't always perfect. ("Yes, you can have a $3,000 study to decide whether people should carry umbrellas in the rain if I can have my $103,000 to study the effects of tequila and gin on sunfish." Someone really should spend a few dollars to study whether that kind of budget process is efficient.)

To keep changes from proliferating wildly and becoming hopelessly convoluted, many projects (particularly large ones) create a *change control board* that reviews and approves (or rejects) change requests. The board should include people who represent the customers ("We really *need* to log in telepathically from home") and the development team ("The best we can do is let you log in on your cell phone").

Even on small projects, it's usually worthwhile to assign someone as the final arbiter. Often that person is either a high-ranking customer (such as the executive champion) or a high-ranking member of the development team (such as the project lead).

During development, it's important to check the documentation to see what you're supposed to be doing. You need to easily find the most recent version of the requirements to see what the application should do. Similarly, you need to find the most recent high-level and low-level designs to see if you're following the plan correctly.

Sometimes, you'll also need to find older versions of the documentation, to find out what changes were made, why they were made, and who made them.

FONT FIASCO

To understand the importance of historical documentation, suppose your application produces quarterly reports showing projected root beer demand. At some point the requirements were changed to require that the report be printed in landscape mode with a 16-point Arial font.

Now suppose you're working on the report to add new columns that group customers by age, weight, ethnic background, car model, and hat size. That's easy enough, but now the report won't fit on the page. If you could bump the font size down to 14-point, everything would fit just fine, but the 16-point Arial requirement is killing you.

At this point, you should go back to the requirements documents and find out why the font requirement was added. If the requirement was added to make it easier to include reports in PowerPoint slides, you may be able to reduce the font size and your boss can live with slightly more crowded slides during his presentations to the VP of Strategic Soft Drink Engineering.

Another option might be to continue producing the original report for presentations and create a new expanded report that includes the new columns for research purposes.

It's even possible that the original issue was that some developers were printing reports with the Comic Sans font. Management didn't think that looked professional enough, so it made a font requirement. They never actually cared about the font size, just the typeface. In that case, you could probably ask to change the requirement again to let you use a smaller font, as long as you stick with Arial.

Unless you have a good document history, you may never know why and when the requirement was changed, so you won't know whether it's okay to change it again.

Meanwhile, as you're changing the font requirement to allow 12-point Arial, one of your coworkers might be changing some other part of the same requirement document (perhaps requiring that all reports must be printed in renewable soy ink on 100% post-consumer recycled paper). If you both open the document at the same time, whichever change is saved second will overwrite the other change, and the first change will be lost. (In programming terms, this is a "race condition" in which the second person wins.)

To prevent this type of conflict, you need a document control system that prevents two people from making changes to the same document at the same time.

To handle all these issues, you need a good document management system. Ideally, the system should support at least the following operations:

- People can share documents so that they can all view and edit them.
- Only one person can edit a document at a given time.
- You can fetch the most recent version of a document.
- You can fetch a specific version of a document by specifying either a date or version number.
- You can search documents for tags, keywords, and anything else in the documents.
- You can compare two versions of a document to see what changed, who changed it, and when the change occurred. (Ideally, you should also see notes indicating why a change was made; although, that's a less common feature.)

Following are some other features that are less common but still useful:

- The ability to access documents over the Internet or on mobile devices.
- The ability for multiple people to collaborate on documents (so they can see each other making changes to a shared document).
- Integration into other tools such as Microsoft Office or project management software.
- Document branches so that you can split a document into two paths for future changes. (This is more useful with program code where you might need to create two parallel versions of the program. Even then it can lead to a lot of confusion.)
- User roles and restricted access lists.
- E-mail change notification.
- Workflow support and document routing.

Some document management systems don't include all these features, and some of these aren't necessary for smaller projects, but they can be nice to have.

The following sections describe some special features of different kinds of documentation that you should save.

HISTORICAL DOCUMENTS

After you've installed some sort of document management system, you may wonder what documents you should put in it. The answer is: ***everything***. Every little tidbit and scrap of intelligence dealing with the project should be recorded for posterity. Every design decision, requirements change, and memo should be tucked away for later use.

If you don't have all this information, it's too easy for project meetings to devolve into finger-pointing sessions and blame-game tournaments. Let's face it; people forget things.

(I'm writing Chapter 2 and I've already forgotten what Chapter 1 was about.) Not every disagreement has the vehemence of a blood feud between vampires and werewolves, but some can grow that bad if you let them. If you have a good, searchable document database, you can simply find the memo where your customer said that all the monitors had to be pink, pick the specific shade, and move on to discuss more important matters.

Collecting every scrap of relevant information isn't quite as big a chore as you might think. Most of the information is already available in an electronic form, so you just need to save it. Whenever someone sends an e-mail about the project, save it. Whenever someone makes a change request, save it. If someone creates a new document and doesn't put it in the document repository, put it there yourself or at least e-mail it to yourself so that there's a record.

The only types of project activity that aren't usually easy to record electronically are meetings and phone calls. You can record meetings and phone calls if you want a record of everything (subject to local law), but on most projects you can just type up a quick summary and e-mail it to all the participants. Anyone who disagrees about what was covered in the meeting can send a follow-up e-mail that can also go into the historical documents.

It's also well worth your effort to thrash through any disagreements as soon as possible, and sending out a meeting summary can help speed that process along. The biggest purpose of documentation is to ensure that everyone is headed in the same direction.

E-MAIL

Memos, discussions about possible change requests, meeting notes, and lunch orders are all easy to distribute via e-mail. Storing those e-mails for historical purposes is also easy: Simply CC a selected e-mail address for every project e-mail. For example, you could create an e-mail address named after the project and copy every project message to that account.

Suppose you're working on project CLASP (CLeverly Acronymed Software Project). Then you would create an e-mail account named CLASP and send copies of any project e-mail to that account.

> **TIP** *I've had project managers who extracted every project e-mail into text files and tucked them away in a folder for later use. That lets you perform all sorts of other manipulations that are tricky inside an e-mail system. For example, you could write a program to search the files for messages from the user Terry that include the words "sick" and "Friday." I've even had project managers who printed out every e-mail; although, that seems a bit excessive. Usually just having the e-mails saved in a project account is good enough.*

Sometimes, it's hard for team members to easily find project-related e-mails in the daily spamalanche of offers for cheap Canadian prescriptions, low interest rates guaranteed by the "U.S. National Bank," letters from your long lost Nigerian uncle, and evacuation notices from your Building Services department.

To make finding project e-mails easier, you can prefix their subjects with an identifier. The following text might show the subject line for an e-mail about the CLASP project.

```
[CLASP] This week's meeting canceled because all tasks are ahead of schedule
```

Of course, if you receive an e-mail with this subject, you should suspect it's a hoax because all tasks have never been ahead of schedule in the entire history of software engineering. I think the day the term "software engineering" was coined, its definition was already a week overdue.

You can further refine the subject identifier by adding classes of messages. For example, `[CLASP.Design]` might indicate a message about design for the CLASP project. You can invent any message classes that you think would be useful. Following is a list of a few that may come in handy.

- `Admin`—Administration
- `Rqts`—Requirements
- `HLDesign`—High-level design
- `LLDesign`—Low-level design
- `Dvt`—Development
- `Test`—Testing
- `Deploy`—Deployment
- `Doc`—Documentation
- `Train`—Training
- `Maint`—Maintenance
- `Wrap`—Wrap-up

> **TIP** *It doesn't matter what subject line tags you use, as long as you're consistent. Make a list at the beginning of the project and make sure everyone uses them consistently.*

You could even break the identifier further to indicate tasks within a message class. For example, the string `[CLASP.LLDesign.1001]` might indicate a message regarding low-level design task 1001.

> **TIP** *Some e-mail systems can even use rules to route particular messages to different folders. For example, the system might be able to copy messages with the word CLASP in the title into a project e-mail folder. (Just don't spend more time programming your e-mail system than on the actual project.)*

If team members use those conventions consistently, any decent e-mail system should make it easy to find messages that deal with a particular part of the project. To find the test messages, you can search for `[CLASP.Test`. To find every CLASP e-mail, search for `[CLASP`.

An alternative strategy is to include keywords inside the message body. You can use a naming convention similar to the one described here, or you can use something more elaborate if you need to. For example, a message might begin with the following text to flag it as involving the testing, bug reports, and login screen.

```
Key: Test

Key: Bugs

Key: Login
```

Now you can search for strings like `Key: Bugs` to find the relevant messages.

In addition to making e-mails easy to find, you should take steps to make them easy to distribute. Create some e-mail groups so that you can distribute messages to the appropriate people. For example, you may want groups for managers, user interface designers, customers, developers, testers, and trainers—and, of course, a group for everyone.

Then be sure you use the groups correctly! Customers don't want to hear the developers argue over whether a b+tree is better than an AVL-tree, and user interface designers don't want to hear the testers dispute the fine points of white-box versus beige-box testing. (In one project I was on, a developer accidentally included customers in an e-mail that described them in less than flattering terms. Basically, he said they didn't really know what they needed. It was true, but they sure didn't like hearing it!)

CODE

Program source code is different from a project's other kinds of documents. Normally, you expect requirements and design documents to eventually stabilize and remain mostly unchanged. In contrast, code changes continually, up to and sometimes even beyond the project's official ending date.

That gives source code control systems a slightly different flavor than other kinds of document control systems. A requirements document might go through a dozen or so versions, but a code module might include hundreds or even thousands of changes. That means the tools you use to store code often don't work particularly well with other kinds of documents and vice versa.

Source code is also generally line-oriented. Even in languages such as C# and Java, which are technically not line-oriented, programmers insert line breaks to make the code easier to read. If you change a line of source code, that change probably doesn't affect the lines around it. Because of that, if you use a source code control system to compare two versions of a code file, it flags only that one line as changed.

In contrast, suppose you added the word "incontrovertibly" to the beginning of the preceding paragraph. That would make every line in the paragraph wrap to the following line, so every line in the paragraph would seem to have been changed. A document revision system, such as those provided by Microsoft Word or Google Docs, correctly realizes that you added only a single word. A source code control system might decide that you had modified every line in the paragraph.

What this means is that you should use separate tools to manage source code and other kinds of documents. This usually isn't a big deal, and it's easy to find a lot of choices online. (In fact, picking one that every developer can agree on may be the hardest part of using a source code control system.)

Ideally, a source code control system enables all the developers to use the code. If a developer needs to modify a module, the system checks out the code to that developer. Other developers can still use the most recently saved version of the code, but they can't edit that module until the first developer releases it. (This avoids the race condition described earlier in this chapter.)

Some source code control systems are integrated into the development environment. They make code management so easy even the most advanced programmers don't mess it up too often.

CODE DOCUMENTATION

Something that most nonprogrammers (and quite a few programmers) don't understand is that code is written for people, not for the computer. In fact, the computer doesn't execute the source code. The code must be compiled, interpreted, and otherwise translated into a sequence of 0s and 1s that the computer can understand.

The computer also doesn't care what the code does. If you tell it to erase its hard disk (something I don't recommend), the computer will merrily try to do just that.

The reason I say source code is written for people is that it's people who must write, understand, and debug the code. The single most important requirement for a program's code is that it be understandable to the people who write and maintain it.

Now I know I'm going to get a lot of argument over that statement. Programmers have all sorts of favorite goals like optimizing speed, minimizing bugs, and including witty puns in the comments. Those are all important, but if you can't understand the code, you can't safely modify it and fix it when it breaks.

Without good documentation, including both design documents and comments inside the code, the poor fool assigned to fix your code will stand little or no chance. This is even more important when you realize that the poor fool may be you. The code you find so obvious and clever today may make no sense at all to you in a year or even a few months. (In some cases, it may not make sense a few hours later.) You owe it to posterity to ensure that your genius is properly understood throughout the ages.

To that end, you need to write code documentation. You don't need to write enormous tomes explaining that the statement `numInvoicesLost = numInvoicesLost + 1` means you are adding 1 to the value `numInvoicesLost`. You can probably figure that out even if you've never seen a line of code before. However, you do need to give yourself and others a trail of breadcrumbs to follow on their quest to figure out why invoices are being sent to employees instead of customers.

Code documentation should include high- and low-level design documents that you can store in the document repository with other kinds of documentation. These provide an overview of the methods the code is using to do whatever it's supposed to do.

Code documentation should also include comments in the code to help you understand what the code is actually doing and how it works. You don't need to comment every line of code (see the

`numInvoicesLost` example again), but it should provide a fairly detailed explanation that even the summer intern who was hired only because he's the boss's nephew can understand. Debugging code should be an exercise in understanding the code and figuring out why it isn't doing what it's supposed to do. It shouldn't be an IQ test.

JBGE

There's a school of thought in software engineering that says you should provide code documentation and comments that are "just barely good enough" (*JBGE*). The idea is that if you provide too much documentation, you end up wasting a lot of time updating it as you make changes to the code.

This philosophy can reduce the amount of documentation you produce, but it's an idea that's easy to take too far. Most programmers like to program (that's why they're not lawyers or doctors) and writing and updating documentation and comments doesn't feel like writing code, so sometimes they skip it entirely.

A software engineering joke says, "Real programmers don't comment their code. If it was hard to write, it should be hard to understand and harder to modify." Unfortunately I've seen plenty of code that proves the connection between poor documentation and difficult modification.

I worked on one project that included more than 55,000 lines of code and fewer than 300 comments. (I wrote a program to count them.) And if there were design documents, I never heard about them. I'm sure the code made sense when it was written, but modifying it was next to impossible. I sometimes spent 4 or 5 days studying the code, trying to figure out how it worked before changing one or two lines. Even after all that time, there was a decent chance I misunderstood something and the change added a new bug. Then I got to remove the change and start over.

I worked on another project that included tons of comments. Probably more than 80 percent of the lines of code included a comment. They were easy to ignore most of the time, but they were always there if you needed them.

After we transferred the project to the company's maintenance organization, the folks in the organization went on a JBGE bender and removed every comment that they felt wasn't absolutely necessary to understand the code. A few months later, they admitted that they couldn't maintain the code because —...drumroll...— they couldn't understand it. In the end, they put all the comments back and just ignored them when they didn't need them.

Yes, excessive code documentation and comments are a hassle and slow you down, so you can't rush off to the next task, but suck it up and write it down while it's still fresh in your mind. You don't need to constantly update your comments every time you change a line of code. Wait until you finish writing and testing a chunk of code. Then write it up and move on with a clear conscience. Comments may slow you down a bit, but I've never seen a project fail because it contained too many comments.

JBGE, REDUX

JBGE is mostly applied to code documentation and comments, but you could apply the same rule to any kind of documentation. For example, you could write barely enough documentation to explain the requirements. That's probably an even bigger mistake than skimping on code comments.

Documentation helps keep the whole project team working toward the same goals. If you don't spell things out unambiguously, developers will start working at cross-purposes. At best you'll lose a lot of time arguing about what the requirements mean. At worst you'll face a civil war that will destroy your team.

As is the case with code documentation and comments, you don't need to turn the requirements into a 1,200-page novel. However, if the requirements are ambiguous or confusing, pull out your thesaurus and clarify them.

JBGE is okay as long as you make sure your documentation actually is GE.

You can extend the JBGE idea even further and create designs that are just barely good enough, write code that's just barely good enough, and perform tests that are just barely good enough. I'm a big fan of avoiding unnecessary work, but if everything you do is just barely good enough, the result probably won't be anywhere near good enough. (Not surprisingly, no one advocates that approach. The JBGE philosophy seems to be reserved only for code comments.)

Some programming languages provide a special kind of comment that is intended to be pulled out automatically and used in text documentation. For example, the following shows a snippet of C# code with XML comments:

```
/// <summary>
/// Deny something bad we did to the media.
/// </summary>
/// <param name="type">What we did (Bug, Virus, PromisedBadFeature, etc.)</param>
/// <param name="urgency">High, Medium, or Low</param>
/// <param name="media">One or more of Blog, Facebook, Friendster, etc.</param>
private void PostDenial(DenialType type, UrgencyType urgency, MediaType media)
{
    . . .
}
```

The comment's `summary` token explains the method's purpose. The `param` tokens describe the method's parameters. The Visual Studio development environment can automatically extract these comments into an XML file that you can then process to produce documentation. The result doesn't explain how the code works, but if you do a good job writing the comments, it does explain the

interface that the method displays to other pieces of code. (Visual Studio also uses these comments to provide help pop-ups called IntelliSense to other developers who call this code.)

As is the case when you write code documentation and other comments, you don't need to constantly update this kind of information as you work on a method. Finish the method, test it, and then write the comments once.

APPLICATION DOCUMENTATION

All the documentation described so far deals with building the application. It includes such items as requirements and design documents, code documentation and comments, meeting and phone call notes, and memos.

At some point you also need to prepare documentation that describes the application. Depending on the eventual number and kinds of users, you may need to write user manuals (for end users, managers, database administrators, and more), quick start guides, cheat sheets, user interface maps, training materials, and marketing materials. You may even need to write meta-training materials to teach the trainers how to train the end users. (No kidding, I've done it.)

In addition to basic printed versions, you may need to produce Internet, screencast, video, and multimedia versions of some of those documents. Producing this kind of documentation isn't all that different from producing requirements documents. You can still store documents in an archive. (Although you may not be able to search for keywords in a video.)

Although creating this material is just another task, don't start too late in the project schedule. If you wait until the last minute to start writing training materials, then the users won't be able to use the application when it's ready. (I remember one project where the requirements and user interface kept changing right up until the last minute. It was somewhat annoying to the developers, but it practically drove our lead trainer insane.)

SUMMARY

Documentation is produced throughout a project's lifespan, starting with early discussions of the project's requirements, extending through design and programming, continuing into training materials, and lasting even beyond the project's release in the form of comments, bug reports, and change requests. To get the most out of your documentation, you need to set up a document tracking system before you start the project. Then you can effectively use the project documents to determine what you need to do and how you should do it. You can also figure out what was decided in the past so that you don't need to constantly rehash old decisions.

Document control is one of the first tools you should set up when you're considering a new project. You can use it to archive ideas before you know what the project will be about or whether there will even be a project. Once you know the project will happen, you should start tracking the project with project management tools. The next chapter describes project management in general and some of the tools you can use to help keep a project moving in the right direction.

EXERCISES

1. List seven features that a document management system should provide.

2. Microsoft Word provides a simple change tracking tool. It's not a full-featured document management system, but it's good enough for small projects. For this exercise, follow these steps:

   a. Create a short document in Word and save it.

   b. Turn on change tracking. (In recent versions of Word, go to the Review tab's Tracking group and click Track Changes.)

   c. Modify the document and save it with a new name. (You should see the changes flagged in the document. If you don't, go to the Review tab's Tracking group and use the drop-down to select Final: Show Markup.)

   d. On the Review tab's Tracking group, click the Reviewing Pane button to display the reviewing pane. You should see your changes there.

   e. In the Review tab's Tracking group, open the Track Changes drop-down and select Change User Name. Change your user name and initials.

   f. Make another change, save the file again, and see how Word indicates the changes.

3. Microsoft Word also provides a document comparison tool. If you followed the instructions in Exercise 1 carefully, you should have two versions of your sample document. In the Review tab's Compare group, open the Compare drop-down and select Compare. (I guess Microsoft couldn't think of synonyms for "compare.") Select the two versions of the file and compare them. How similar is the result to the changes shown by change tracking? Why would you use this tool instead of change tracking?

4. Like Microsoft Word, Google Docs provides some simple change tracking tools. Go to `http://www.google.com/docs/about/` to learn more and to sign up. Then create a document, save it, close it, reopen it, and make changes to it as you did in Exercise 1.

   To view changes, open the File menu and select See revision history. Click the See more detailed revisions button to see your changes.

5. What does JBGE stand for and what does it mean?

▸ WHAT YOU LEARNED IN THIS CHAPTER

- Documentation is important at every step of the development process.
- Good documentation keeps team members on track, provides a clear direction for work, and prevents arguments over issues that were previously settled.
- Document management systems enable you to:
  - Share documents with other team members.
  - Fetch a document's most recent version.
  - Fetch an earlier version of a document.
  - Search documents for keywords.
  - Show changes made to a document.
  - Compare two documents to show their differences.
  - Edit a document while preventing someone else from editing the document at the same time.
- A simple way to store project history is to create an e-mail account named after the project and then send copies of all project correspondence to that account.
- You can use e-mail subject tags such as `[CLASP.Rqts]` to make finding different types of project e-mails easy.
- Types of documentation may include:
  - Requirements
  - Project e-mails and memos
  - Meeting notes
  - Phone call notes
  - Use cases
  - High-level design documents
  - Low-level design documents
  - Test plans
  - Code documentation
  - Code comments
  - Extractable code comments
  - User manuals
  - Quick start guides
  - Cheat sheets

- User interface maps
- Training materials
- Meta-training materials
- Marketing materials

- JBGE (Just Barely Good Enough) states that you should provide only the absolute minimum number of comments necessary to understand the code.

3

Project Management

Effective leadership is putting first things first. Effective management is discipline, carrying it out.

—Stephen Covey

WHAT YOU WILL LEARN IN THIS CHAPTER:

- What project management is and why you should care
- How to use PERT charts, critical path methods, and Gantt charts to create project schedules and estimate project duration
- How you can improve time estimates
- How risk management lets you respond quickly and effectively to problems

Part of the reason you implemented all the change tracking described in the preceding chapter is so that you have historical information when you're writing your memoirs. It's so you know what happened, when, and why.

In addition to this peek into the past, you also need to keep track of what's going on in real time. Someone needs to track what's happening, what should be happening, and why the two don't match. That's where project management comes in.

Many software developers view management with suspicion, if not downright fear or loathing. They feel that managers were created to set unrealistic goals, punish employees when those goals aren't met, and take credit if something accidentally goes right. There are certainly managers like that, and Scott Adams has made a career out of making fun of them in his *Dilbert* comic strip, but some management is actually helpful for producing good software.

Management is necessary to ensure that goals are set, tracked, and eventually met. It's necessary to keep team members on track and focused on the problems at hand, without becoming distracted by inconsequential side issues such as new unrelated technology, impending layoffs, and *Angry Birds* tournaments.

On smaller projects, a single person might play multiple management roles. For example, a single technical manager might also handle project management tasks. On a really small project, a single person might perform every role including those of managers, developers, testers, and end users. (Those are my favorite kinds of projects because the meetings and arguments are usually, but not always, short.)

No matter how big the project is, however, management tasks must be performed. The following sections describe some of the key management responsibilities that must be handled by someone for any successful software development project.

EXECUTIVE SUPPORT

Lack of executive management support is often cited as one of the top reasons why software projects fail. This is so important, it deserves its own note.

> **NOTE** *To be successful, a software project must have consistent executive management support.*

The highest-ranking executive who supports your project is often known as an ***executive champion*** or an ***executive sponsor***.

Robust executive support ensures that a project can get the budget, personnel, hardware, software, and other resources it needs to be successful. It lets the project team specify a realistic schedule even when middle management or customers want to arbitrarily shorten the timeline. Managers with limited software development experience often don't understand that writing quality software takes time. The end result isn't physical, so they may assume that you can compress the schedule or make do with fewer resources if you're properly motivated by pithy motivational slogans.

Unfortunately, that usually doesn't work well with software development. When developers work long hours for weeks at a time, they burn out and write sloppy code. That leads to more bugs, which slows development, resulting in a delayed schedule, greater expense, and a low-quality result. Not only do you fail to achieve the desired time and cost-savings, but also you get to take the blame for missing the impossible deadlines.

Executive support is also critical for allowing a project to continue when it encounters unexpected setbacks such as missed deadlines or uncooperative software tools. In fact, unexpected setbacks are so common that you should expect some to occur, even if you don't know what they will be. (Donald Rumsfeld would probably consider them "known unknowns.")

Overall the executive champion provides the gravitas needed for the project to survive in the rough-and-tumble world of corporate politics. It's a sad truth that different parts of a company don't

always have the same goals. (In management-speak, you might say the oars aren't all pulling in the same direction.) The executive champion can head off attempts to cancel a project and transfer its resources to some other part of the company.

In cases like those, work can be somewhat unnerving, even if you do have strong executive support. I once worked on a project where both our executive champion and our arch nemesis were corporate vice presidents directing thousands of employees. At times I felt like a movie extra hoping Godzilla and Mothra wouldn't step on us while they slugged it out over Japan. After two years of unflagging support by our champion, we finished the project and transferred it to another part of the company where it was quite successful for many years.

Executive champion duties include:

- Providing necessary resources such as budgets, hardware, and personnel
- Making "go/no-go" decisions and deciding when to cancel the project
- Giving guidance on high-level issues such as how the project fits into the company's overall business strategy
- Helping navigate any administrative hurdles required by the company
- Defining the business case
- Working with users and other stakeholders to get buy-in
- Providing feedback to developers about implemented features
- Buffering the project from external distractions (such as the rest of the company)
- Refereeing between managers, users, developers, and others interested in the project
- Supporting the project team
- Staying out of the way

The last point deserves a little extra attention. Most executives are too busy to micromanage each of the projects they control, but this can sometimes be an issue, particularly if the executive champion is near the bottom of the corporate ladder. If you are an executive champion, monitor the project to make sure it's headed in the right direction and that it's meeting its deadlines and other goals, but try not to add extra work. As Tina Fey says in her book *Bossypants*, "In most cases being a good boss means hiring talented people and then getting out of their way."

However, studies have shown that more engaged executives result in more successful projects, so don't just get things started and then walk away.

PROJECT MANAGEMENT

A *project manager* is generally the highest-ranking member of the project team. Ideally, this person works with the team through all stages of development, starting with requirements gathering, moving through development and testing, and continuing until application rollout (and sometimes even beyond into future versions).

The project manager monitors the project's progress to ensure that work is heading in the right direction at an acceptable pace and meets with customers and other stakeholders to verify that the finished product meets their requirements. If the development model allows changes, the project manager ensures that changes are made and tracked in an organized manner so that they don't get lost and don't overwhelm the rest of the team.

A project manager doesn't necessarily need to be an expert in the users' field or in programming. However, both of those skills can be extremely helpful because the project manager is often the main interface between the customers and the rest of the project team.

Project manager duties include:

- Helping define the project requirements
- Tracking project tasks
- Responding to unexpected problems
- Managing risk
- Keeping users (and the executive champion) up-to-date on the project's progress
- Providing an interface between customers and developers
- Managing resources such as time, people, budget, hardware, and software tools
- Managing delivery

MYRIAD MANAGERS

There are a lot of kinds of project managers in addition to software project managers. Construction, architecture, engineering, and other fields have project managers. Just about any activity that involves more than a few people needs someone to perform project management duties, even if that person isn't called a project manager.

This book even has a project manager extraordinaire, Adaobi Obi Tulton; although her title is project editor. She makes sure I'm turning chapters in on time, passes chapters to various technical and copy editors, and generally guides the book during its development.

In practice, some project managers are promoted from the developer ranks, so they often have good development skills but weak project management skills. They can give useful advice to other programmers about how a particular piece of code should be written or how one subsystem should interact with another. They can provide technical leadership, but they're not always good at recognizing and handling scheduling problems when something goes wrong.

For that reason, some larger projects divide the project manager's duties among two or more people. One project I worked on had a person dedicated to task tracking and making sure we kept to the schedule. She had training in project management but no programming experience. If something

started to slip, she immediately jumped all over the issue, figured out how much delay was required, asked about contingencies in case the task couldn't be finished, determined whether the delay would affect other tasks, and did all the nontechnical things a project manager must handle.

That person was called the "project manager," in contrast with the other project manager who was called the "project manager." It got a little confusing at times. Perhaps we should have called the task-tracking person the "developer babysitter" or the "border collie" because she gently guided us toward our goals by nipping at our heels. Often people call the other manager the "technical project manager"; although, that person may also handle nontechnical tasks such as interaction with customers and executives.

Meanwhile the "main" project manager was freed up to attack the problem from the development side. He could work with the developers to figure out what was wrong and how to fix it.

When I first encountered this set up, I thought it was kind of silly. Couldn't a single project manager handle both technical and tracking tasks? In our project, the separation actually made things easier. This may not be the right approach for every project, particularly small ones, but it was useful in our case.

If you are a project manager or want to become one, you should do a lot more reading about specific tools and techniques that are useful for keeping a project on track.

Before moving on to other topics, however, I want to cover a few more project management issues in greater detail. The next three sections describe PERT charts, the critical path method (CPM), and Gantt charts. PERT charts and CPM are generally used together but are separated here so that you can digest them in smaller pieces. Together these three tools can help you study the project's total duration, look for potential bottlenecks, and schedule the project's tasks.

However, you can't understand how tasks fit into a schedule unless you know how long those tasks will take, so the last sections about project management deal with predicting task lengths and with risk management.

PERT Charts

A *PERT chart* (PERT stands for Program Evaluation and Review Technique) is a graph that uses nodes (circles or boxes) and links (arrows) to show the precedence relationships among the tasks in a project. For example, if you're building a bunker for use during the upcoming zombie apocalypse, you need to build the outer defense walls before you can top them with razor wire.

PERT charts were invented in the 1950s by the United States Navy. They come in two flavors: activity on arrow (AOA), where arrows represent tasks and nodes represent milestones and activity on node (AON), where nodes represent tasks and arrows represent precedence relations. Activity on node diagrams are usually easier to build and interpret, so that's the kind described here.

To build an AON PERT chart, start by listing the tasks that must be performed, the tasks they must follow (their predecessors), and the time you expect each task to take. (You can also add best-case and worst-case times to each task if you want to perform more extensive analysis of the tasks and what happens when things go wrong.)

Note that you don't need to include every possible combination of predecessors. For example, suppose task C must come after task B, which must come after task A. In that case, task C must come after

task A, but you don't need to include that relationship in the table if you don't want to. The fact that task C must come after task B is enough to represent that relationship. However, you also don't need to remove every unnecessary relationship. Those extra relationships won't hurt anything.

If you like, you can add a Start task as a predecessor for any other tasks that don't have predecessors. Similarly, you can add a Finish task for any other tasks that don't have successors.

To make rearranging tasks easy, make an index card or sticky note for each task. (You can draw the chart on a piece of paper or with a drawing tool, but index cards and sticky notes make it easy to shuffle tasks around if necessary.) Include each task's name, predecessors, and expected time.

Then to build the chart, follow these steps:

1. Place the Start task in a Ready pile. Place the other tasks in a Pending pile.
2. Position the tasks in the Ready pile in a column to the right of any previously positioned tasks. (The first time through, the Ready pile only contains the Start task, so position it on the left side of your desk.)
3. Look through the tasks in the Pending pile and cross out the predecessors that you just positioned. (Initially that means you'll be crossing out the Start task.) If you cross out a card's last predecessor, move it to the Ready pile.
4. Return to step 2 and repeat until you have positioned the Finish task.

PUZZLING PREDECESSORS

If you don't move any tasks into the Ready pile during step 3, that means the tasks have a predecessor loop. For example, task A is task B's predecessor and task B is task A's predecessor.

For example, at my college, you needed to pay registration fees before you could get your student ID; you needed a student ID to get financial aid checks; and you needed financial aid checks to pay registration fees. (At least, you probably do if you need financial aid.) You needed to fill out extra paperwork to break out of the predecessor loop.

After you finish positioning all of the cards, draw arrows representing the predecessor relationships. (You may want to use a dry-erase marker so that you can get the arrows off your desk later.)

At this point, you have a chart showing the possible paths of execution for the tasks in the project.

EXAMPLE Building a PERT Chart

The steps for building a PERT chart are a bit confusing, so let's walk through an example that creates a PERT chart for a project that builds a bunker to protect you and your video games in case of a zombie apocalypse. (The U.S. Strategic Command actually developed a plan for fighting off a zombie apocalypse as part of a training exercise. You can read it at `i2.cdn.turner.com/cnn/2014/images/05/16/dod.zombie.apocalypse.plan.pdf`.)

Start by building a table that lists the tasks, their predecessors, and the times you expect them to take. Table 3-1 shows some of the tasks you would need to perform to build the bunker. To keep things simple, I've omitted a lot of details such as installing sewer lines, building forms for pouring concrete, and obtaining permits (assuming the planning officials haven't been eaten yet).

TABLE 3-1: Tasks for a Zombie Apocalypse Bunker

TASK	TIME (DAYS)	PREDECESSORS
A. Grade and pour foundation.	5	—
B. Build bunker exterior.	5	A
C. Finish interior.	3	B
D. Stock with supplies.	2	C
E. Install home theater system.	1	C
F. Build outer defense walls.	4	—
G. Install razor wire on roof and walls.	2	B, I
H. Install landmines (optional).	3	—
I. Install surveillance cameras.	2	B, F

After you've built the task table, create index cards for the tasks (or be prepared to draw them with a drawing tool). Figure 3-1 shows what the card for task I might look like.

FIGURE 3-1: Each task's card should hold its name, duration, and predecessors. You'll fill in the total time later.

Next, start working through the four steps described earlier to arrange the cards. This is a lot easier to understand if you go to the trouble of creating index cards or sticky notes instead of trying to imagine what they would look like. Trust me. If you found the steps confusing, make the cards.

1. Place the Start task in a Ready pile. Place the other tasks in a Pending pile.

   Figure 3-2 shows the initial positions of the cards. (I've omitted the task names and abbreviated a bit to save space.)

2. Position the tasks in the Ready pile in a column to the right of any previously positioned tasks. (The first time through, the Ready pile contains only the Start task. Just position it on the left side of your desk.)

3. Look through the tasks in the Pending pile and cross out the predecessors that you just positioned. (Initially, that means you'll be crossing out the Start task.) If you cross out a card's last predecessor, move it to the Ready pile.

   Referring to Figure 3-2, you see that tasks A, F, and H have the Start task as predecessors. In fact, the Start task is the only predecessor for those tasks, so when you cross out the Start task, you move tasks A, F, and H into the Ready pile. Figure 3-3 shows the new arrangement.

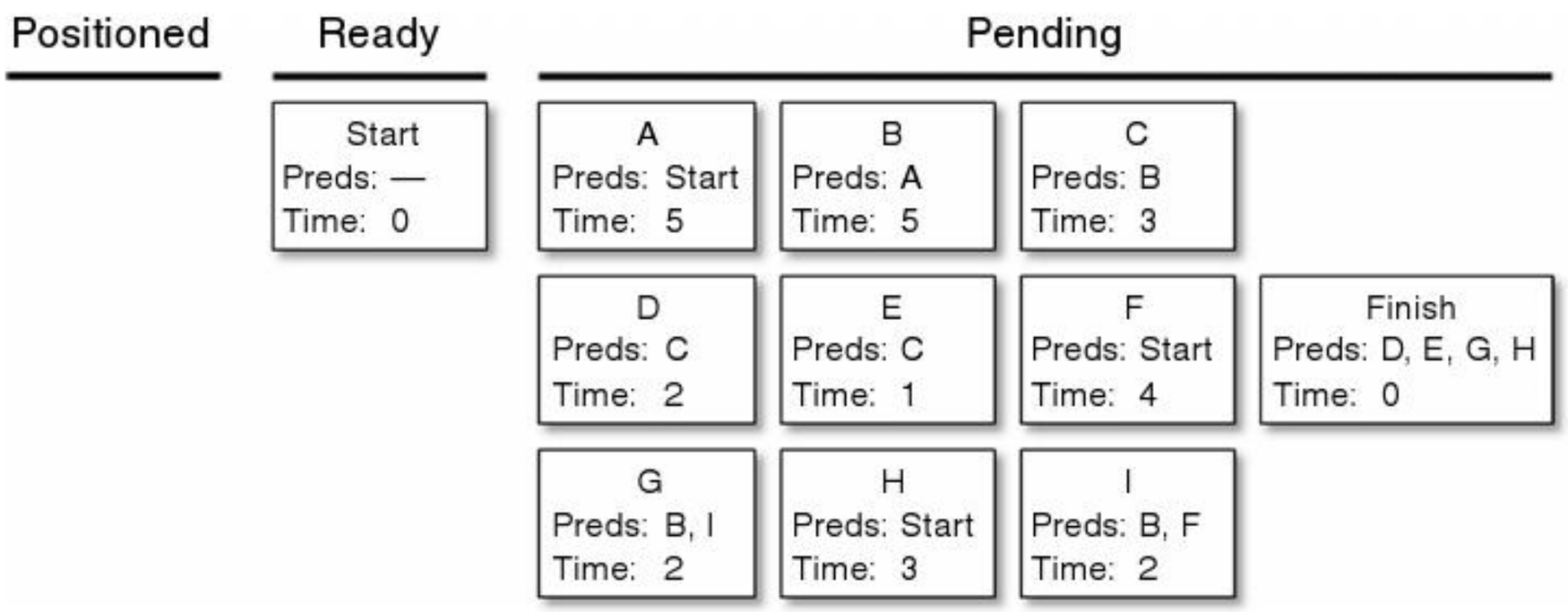

FIGURE 3-2: Initially only the Start task is in the Ready pile.

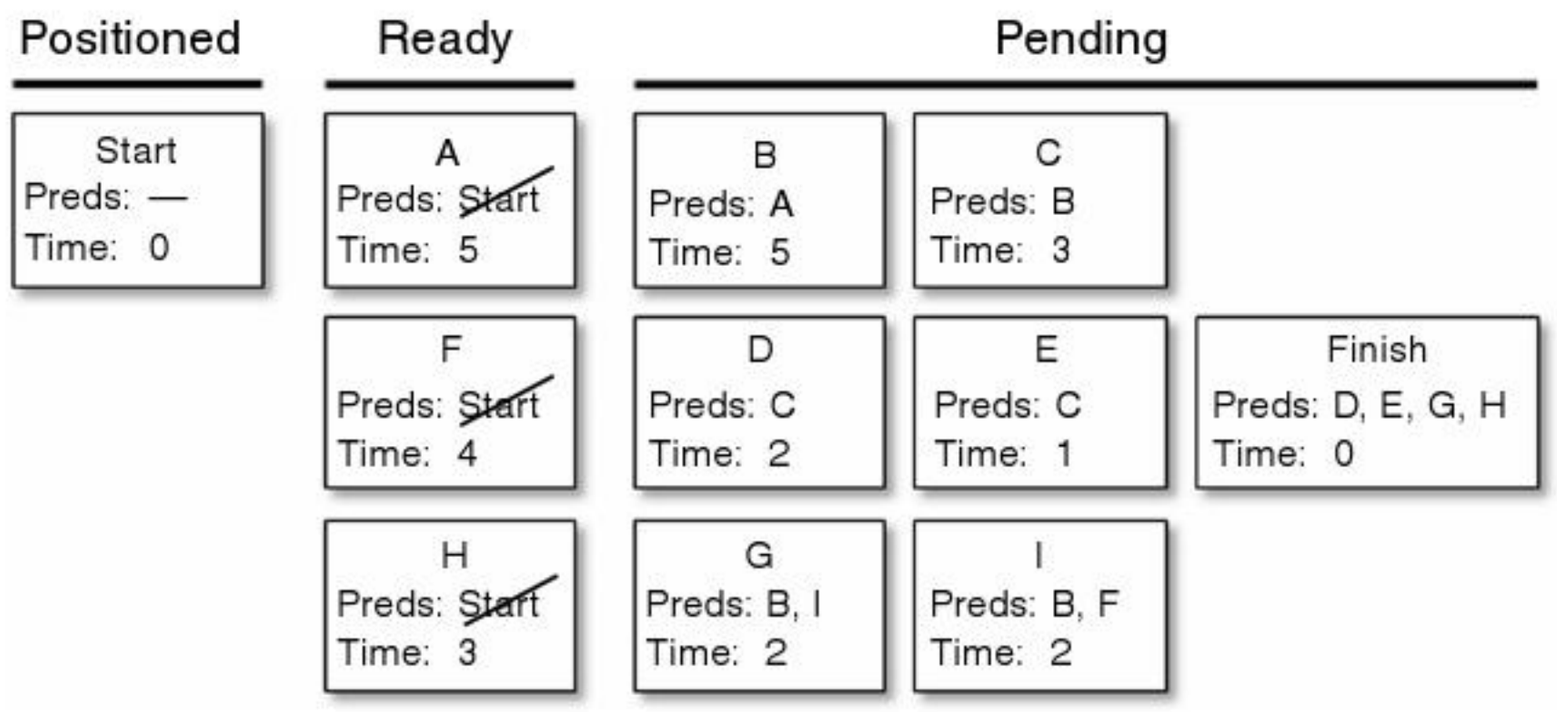

FIGURE 3-3: After one round, the Start task is positioned and tasks A, F, and H are in the Ready pile.

4. Return to step 2 and repeat until you have positioned the Finish task.

   To do that, position tasks A, F, and H because they're in the Ready pile. Then cross them out for any tasks that are still in the Pending pile. When you cross out those tasks, task B loses its last predecessor so move it into the Ready pile. Figure 3-4 shows the new arrangement.

5. Return to step 2 and repeat until you have positioned the Finish task.

   This time position task B and remove it from the remaining tasks' predecessor lists. After you cross task B off, tasks C and I have no more predecessors so move them to the Ready pile. Figure 3-5 shows the new arrangement.

   By now you probably have the hang of it. Position tasks C and I, and remove them from the Pending tasks' predecessor lists. That removes the last predecessors from tasks D, E, and G, so move them to the ready pile, as shown in Figure 3-6.

In the next round, position tasks D, E, and G, and move the Finish task to the Ready pile. Then one final round positions the Finish task.

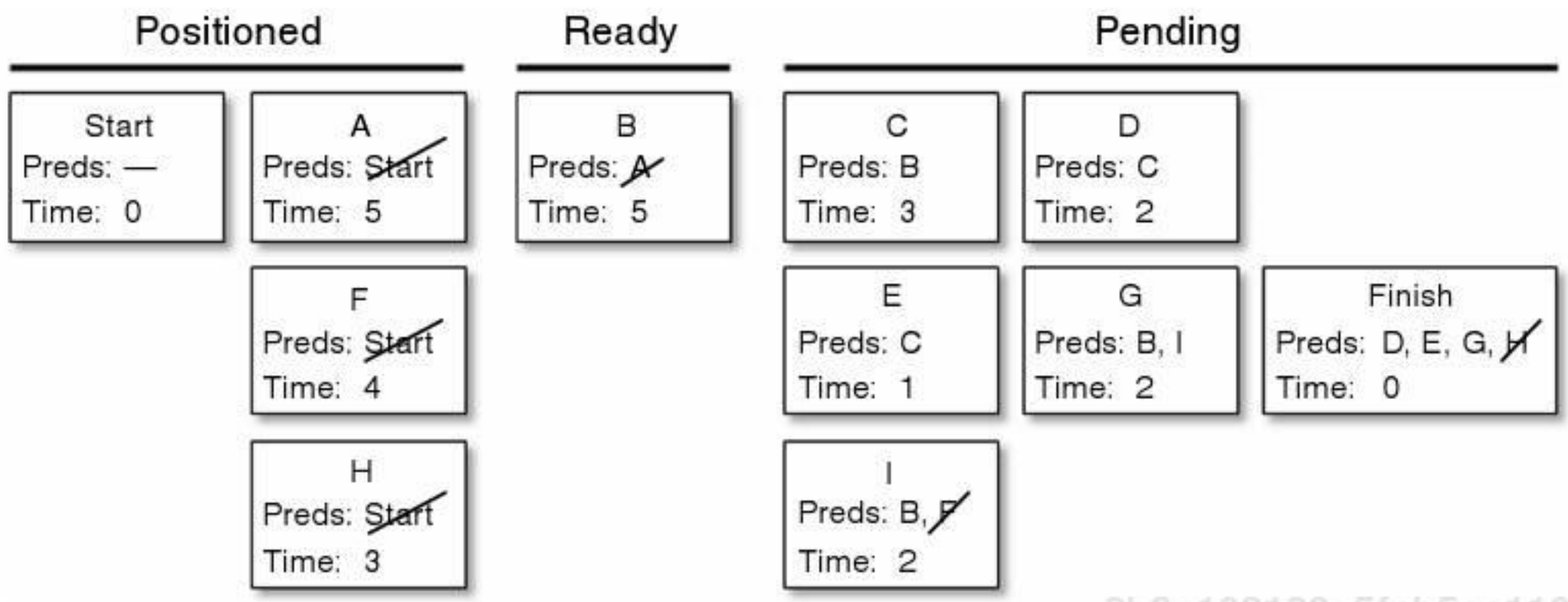

FIGURE 3-4: After two rounds, the Start task and tasks A, F, and H are positioned. Task B is in the Ready pile.

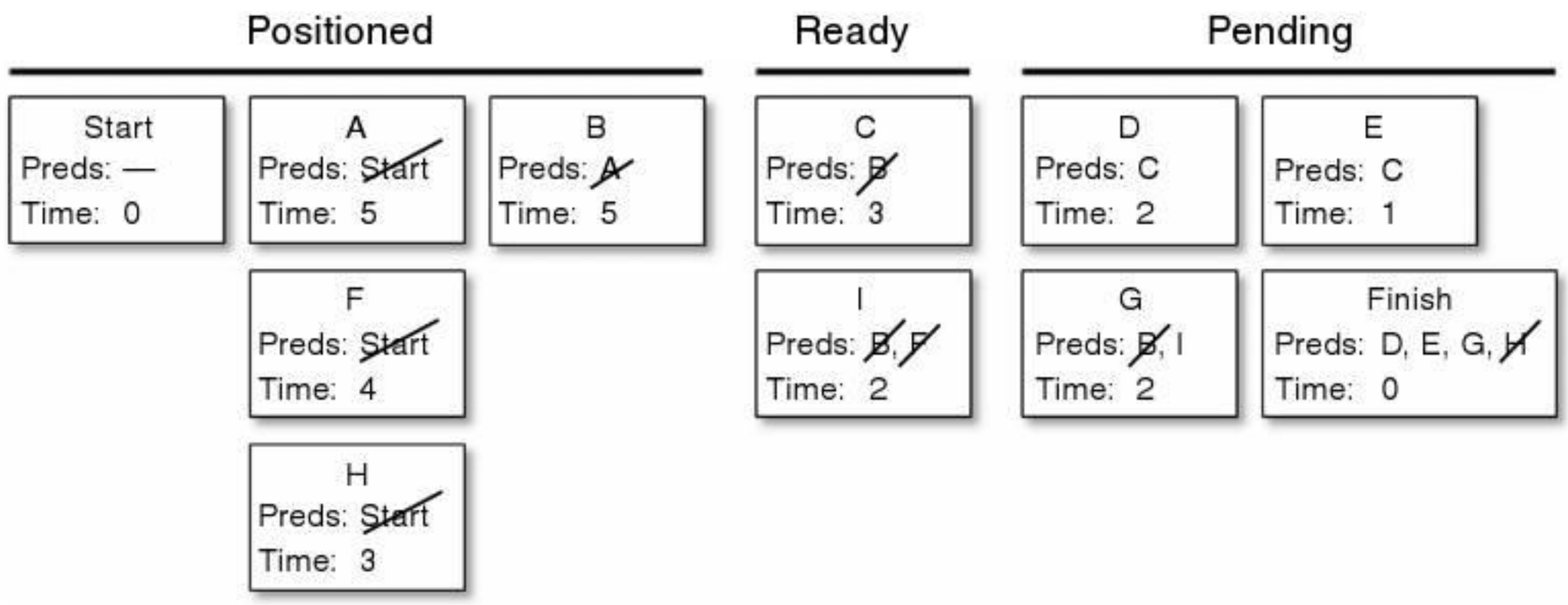

FIGURE 3-5: After three rounds, the Start task and tasks A, F, H, and B are positioned. Tasks C and I are in the Ready pile.

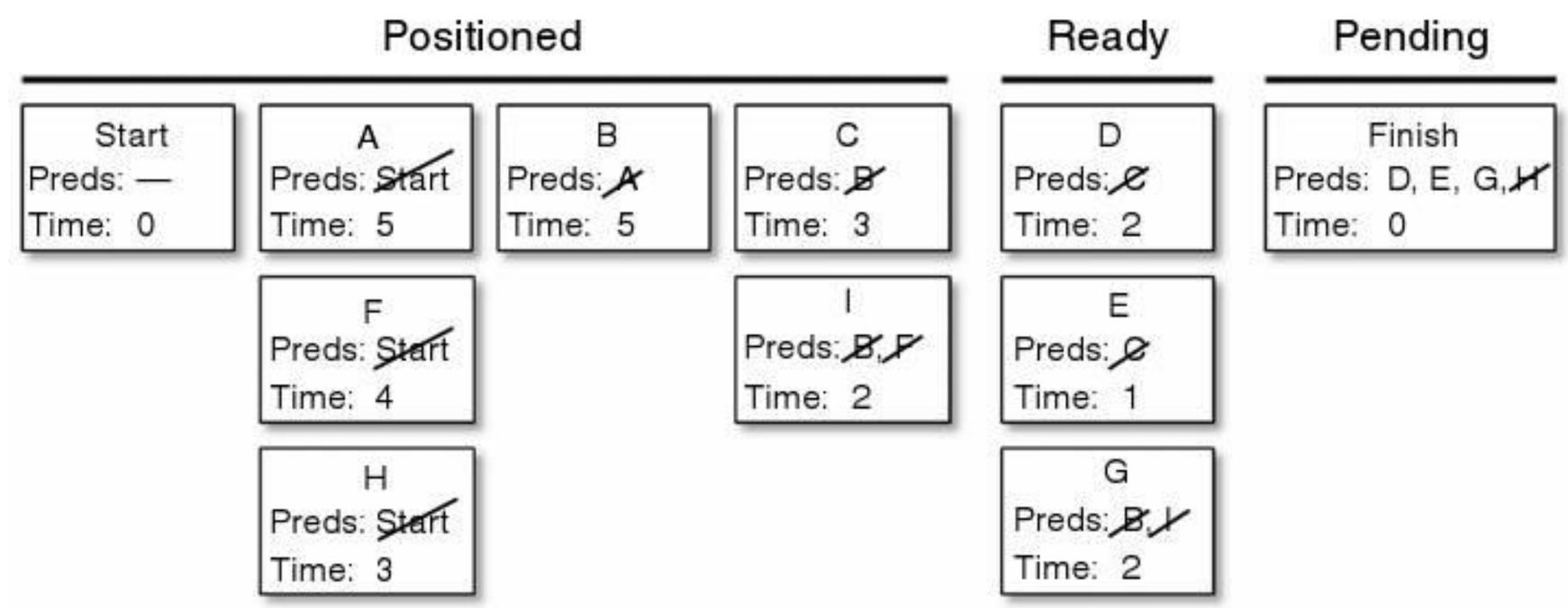

FIGURE 3-6: After four rounds, only the Finish task is still in the Pending pile.

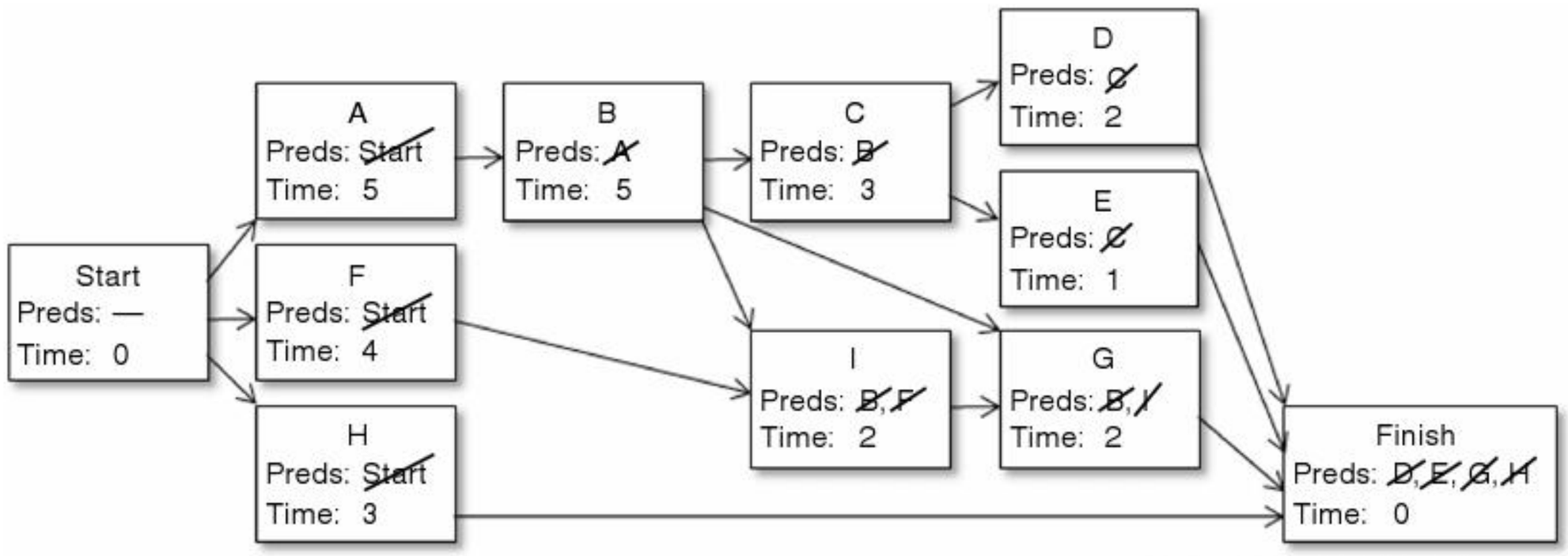

FIGURE 3-7: This PERT chart shows the paths of execution of the project's tasks.

Now draw arrows showing the predecessor relationships between the tasks. You may need to adjust the spacing and vertical alignment of the tasks to make the arrows look nice. Figure 3-7 shows the final result.

To check your work, you can verify that each task has one arrow entering it for each of its predecessors. For example, task G has two predecessors, so it should have two arrows entering it.

Critical Path Methods

PERT charts are often used with the critical path method, which was also invented in the 1950s. That method lets you find critical paths through the network formed by a PERT chart.

A *critical path* is a longest possible path through the network. (I say "a longest" instead of "the longest" because there may be more than one path with the same longest length.)

For example, refer to the PERT network shown in Figure 3-7. The path Start➢H➢Finish has a total time of 3 days. (No charge for the Start and Finish, plus 3 days for task H.)

Similarly, the path Start➢F➢I➢G➢Finish has a total time of 0 + 4 + 2 + 2 + 0 = 8 days.

With a little study of Figure 3-7 and some trial and error, you can determine that this network has a single longest path: Start➢A➢B➢C➢D➢ Finish with a total length of 0 + 5 + 5 + 3 + 2 + 0 = 15 days. Because that's the longest path, it is also the critical path.

If any task along the critical path is delayed, the project's final completion is also delayed. For example, if task C "Finish the interior" takes 5 days instead of 3 (perhaps you decided to add a nice bar with beer taps), then the whole project will take 17 days instead of 15.

For a simple project like this one, it's fairly easy to find the critical path. For projects containing hundreds or even thousands of tasks, this could be a lot harder. Fortunately, there's a relatively easy way to find critical paths.

Start at the left with the Start task. It takes no time to start, so label that task with the total time 0.

Now move to the right, one column at a time. For each task in the current column, set its total time equal to that task's time plus the largest total time for its predecessor tasks.

While you're at it, highlight the link that came from the predecessor with the greatest total cost. If more than one predecessor is tied for the largest total time, highlight them both.

When you're done, the Finish task will be labeled with the total time to complete the project (assuming nothing goes wrong, of course). You can follow the highlighted links back through the network to find the critical paths.

EXAMPLE Critical Paths

In this example, let's walk through the steps for adding total time and critical path information to the zombie apocalypse bunker project PERT chart shown in Figure 3-7.

Start by setting the total time for the Start task to 0.

Referring to Figure 3-7, you can see that the next column of tasks holds tasks A, F, and H, which have expected times 5, 4, and 3, respectively. Each has only Start as a predecessor, and that task has a total time 0 (we just labeled it), so each of these tasks' total time is the same as its own expected time. (So far, not too interesting.)

The next column holds only task B. It has an expected time of 5 and a single predecessor with time 5, so its total time is 5 + 5 = 10. Figure 3-8 shows the network at this point. The new total times and the selected links are highlighted in bold.

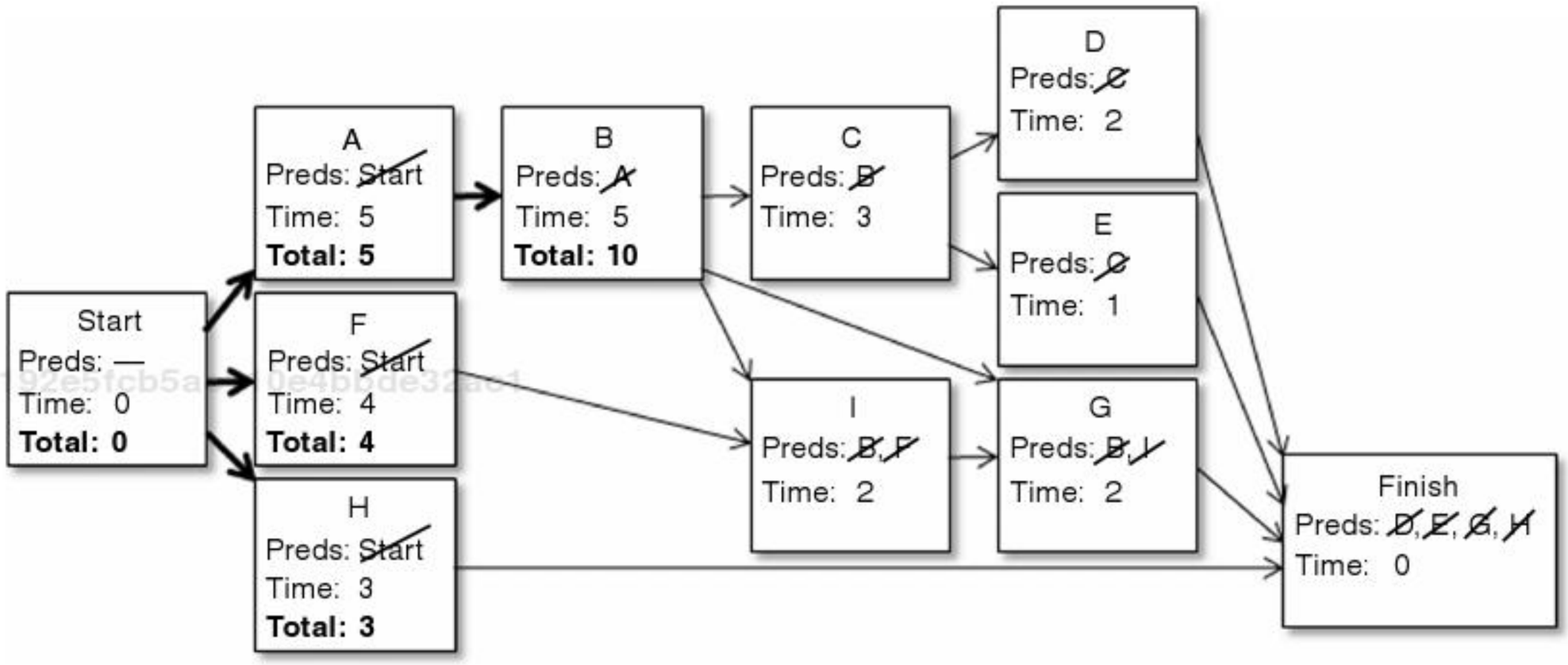

FIGURE 3-8: The total time for each task is its expected time plus the largest of its predecessors' total times.

Now things get a bit more interesting. The next column holds tasks C and I. Task C has an expected time of 3 and a single predecessor with a total time of 10, so its total time is 3 + 10 = 13.

Task I has an expected time of 2. It has two predecessors with total times of 10 and 4, so its total time is 2 plus the larger of 10 and 4 or 2 + 10 = 12. Figure 3-9 shows the updated network.

The next column holds tasks D, E, and G.

Task D has an expected time of 2. Its single predecessor, C, has a total time of 13, so task D's total time is 2 + 13 = 15.

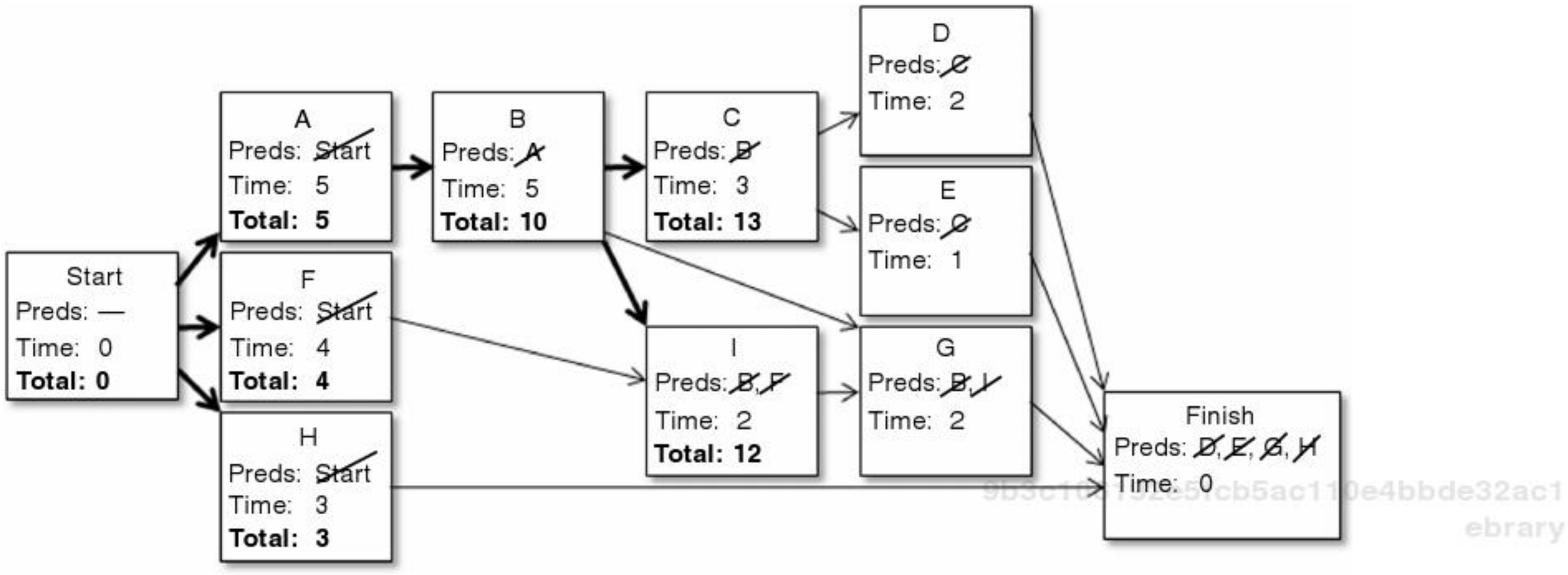

FIGURE 3-9: Task I's largest time predecessor is task B, so task I has a total time of 2 + 10 = 12.

Task E has an expected time of 1. It also has the predecessor C with the total time 13, so task E's total time is 1 + 13 = 14.

Task G has an expected time of 2. It has two predecessors: B with a total time of 10 and I with a total time of 12. That means task G's total time is 2 + 12 = 14.

The final column holds the Finish task. It has an expected time of 0, so its total time is the same as its predecessor with the largest total time. That's task D with a total time of 15. Figure 3-10 shows the final network.

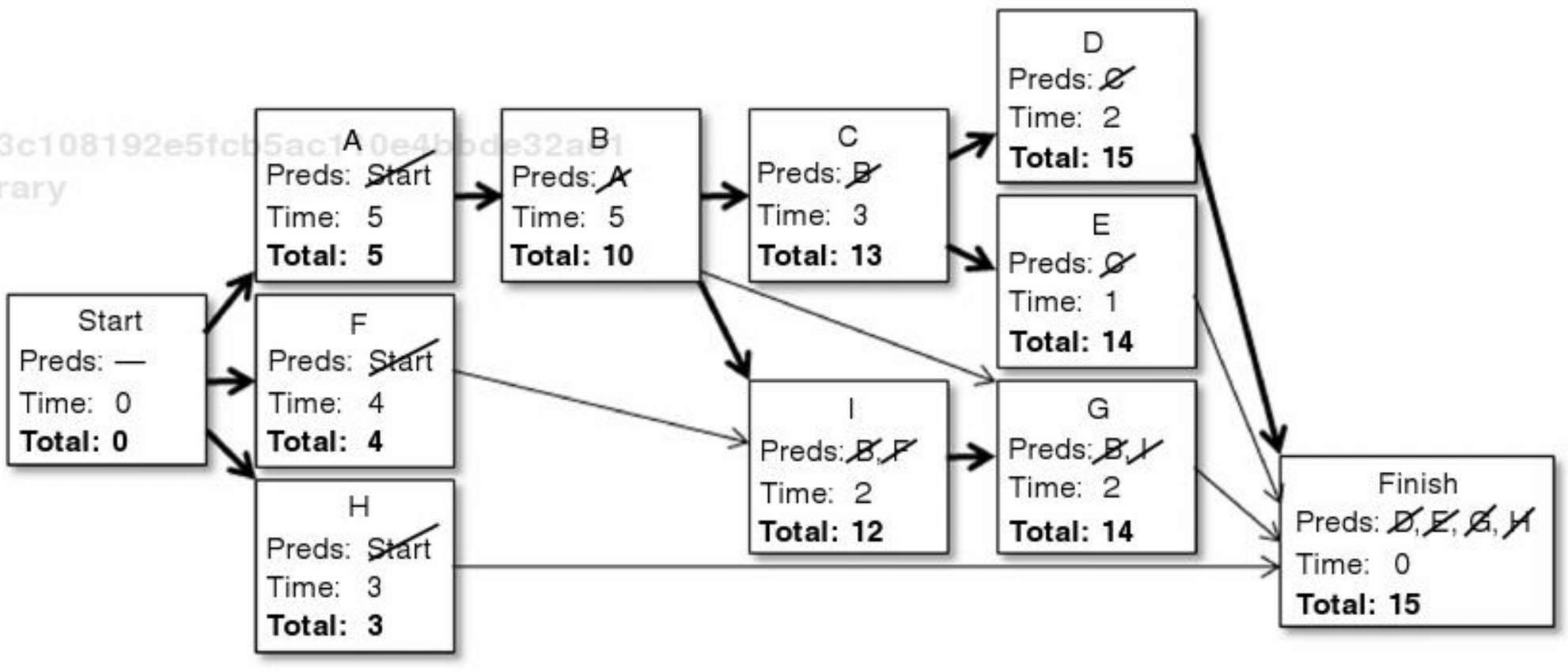

FIGURE 3-10: The complete zombie apocalypse bunker project has a total time of 15 days.

You can trace the bold arrows backward from the Finish to the Start in Figure 3-10 to find the critical path. Those tasks are (in their forward order) Start➢A➢B ➢C➢D➢Finish (as we found earlier).

In addition to showing you the critical path, the PERT network with total times can help you study the project for other possible problems. For example, Figure 3-10 holds two "almost critical paths." The paths through tasks E and G to the Finish task have total times of 14 days, which is only 1 day less than the true critical path. That means if any tasks along those paths are delayed by more than 1 day, the project's completion will be delayed.

The finished network also shows that Tasks F and H don't play a major role in the project's final completion time. Task F could stretch out for up to 10 days without changing the critical path. Task H could run even longer, lasting up to 15 days without impacting the finish date.

To look at this another way, that means you have some flexibility with tasks F and H. You can delay their start a bit if you want without changing the critical path. Sometimes, delaying a task can be useful to balance staffing levels. (After they finish building the bunker in task B, you may want to use the same masons to build the outer defense walls in task F.)

There may also be some reason to rearrange tasks slightly. For example, task H is a landmine installation. The whole project will probably be a lot safer if you delay that as long as possible so that people working on the project don't need to worry about stepping in the wrong places.

Gantt Charts

A *Gantt chart* is a kind of bar chart invented by Henry Gantt in the 1910s to show a schedule for a collection of related tasks. The fact that we're still using them more than 100 years later shows how useful they are for project scheduling.

A Gantt chart uses horizontal bars to represent task activities. The bars' lengths indicate the tasks' durations. The bars are placed horizontally on a calendar to show their start and stop times. Arrows show the relationships between tasks and their predecessors much as they do in a PERT chart.

Figure 3-11 shows a Gantt chart for the zombie apocalypse bunker project that I drew in Microsoft Excel. I've followed a common practice and repeated the task names on the right to make it easier to see which tasks start the arrows. Arrows lead from the end of each task to the beginning of successor tasks.

	A	B	C
1			
2	Task	Time	Preds
3	Start	0	—
4	A. Grade and pour foundation	5	Start
5	B. Build bunker exterior	5	A
6	C. Finish interior	3	B
7	D. Stock with supplies	2	C
8	E. Install home theater system	1	C
9	F. Build outer defense walls	4	Start
10	G. Install razor wire on roof and walls	2	B, I
11	H. Install landmines (optional)	3	Start
12	I. Install surveillance cameras	2	B, F
13	Finish	0	D, E, G, H

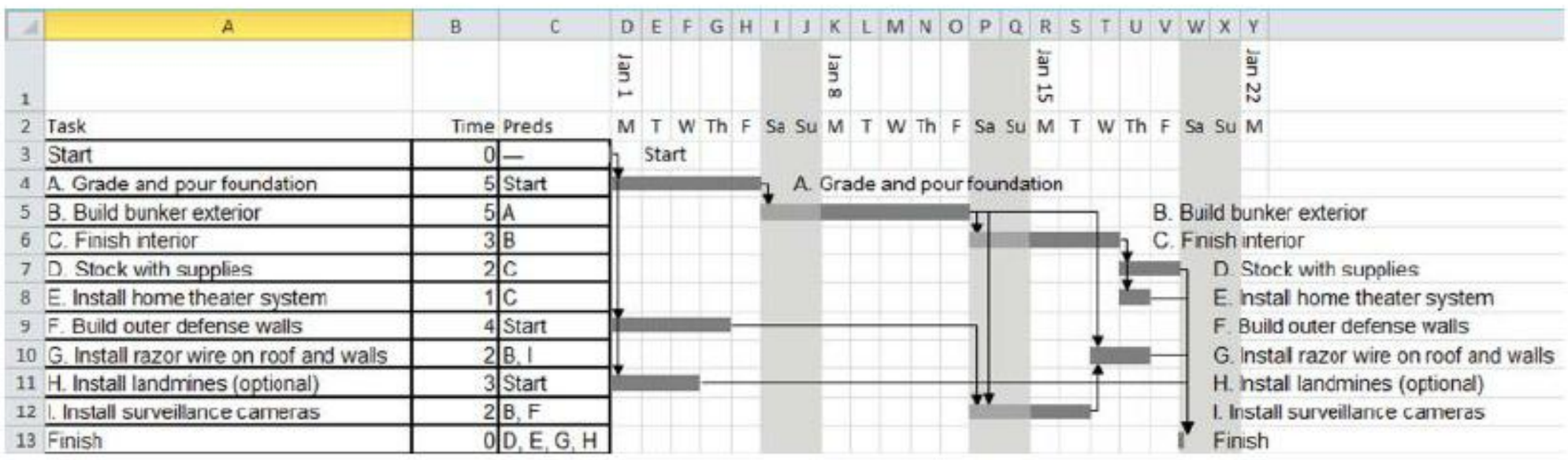

FIGURE 3-11: A Gantt chart shows task durations, start times, end times, and dependencies.

Notice that some of the tasks have been extended to cover the gaps created by weekends. (Figure 3-11 ignores holidays such as New Year's Day for simplicity, but in a real project you would need to account for them.) Also notice that the weekends have extended the project's total duration from 15 working days to 19 calendar days. (In case of a real zombie emergency, you might want to work through the weekends and holidays. I'm sure the zombies will!)

To build a Gantt chart, list the tasks, their durations, and their predecessors on the left, as shown in Figure 3-11.

Next, cut out a thin rectangle for each task. Give each rectangle a width that represents its duration. (For example, you could make each rectangle 1-inch wide per day.) Write the tasks' names on their rectangles.

Next, move from left to right through the columns in the PERT chart, placing each rectangle so that its left edge lines up with the right edge of its rightmost predecessor rectangle. For example, in Figure 3-11 the left edge of task G lines up with the right edge of task I.

If a rectangle includes a weekend, lengthen it so that it gets its required number of working days.

Finally, after you've positioned all the rectangles, add arrows to show the predecessor relationships.

Scheduling Software

The preceding sections explained how you can build PERT charts, find critical paths, and draw Gantt charts by hand. The process isn't too difficult, but the result isn't flexible. For example, suppose you decide to add a gas-powered generator and you want to run underground cables between it and the bunker before you pour the foundation.

Or suppose you start work and building the bunker takes longer than expected. In both of those cases, you need to shift some of the tasks farther to the right. Because I drew the schedule by hand, rearranging those tasks can be a hassle. The problem would be much worse for larger projects.

Fortunately, there are lots of project scheduling tools available for your computer. They make building schedules relatively easy and provide lots of extra features. For example, some enable you to click and drag to connect two tasks or to change a task's duration.

Some of those tools also enable you to define other kinds of relationships between tasks. For example, you might indicate that two tasks should start at the same time or that one task should start five days after another task starts.

If you need to manage a lot of project schedules or schedules with many tasks, you should try some of these tools to find one you like.

Predicting Times

PERT charts, critical path methods, and Gantt charts are great tools for figuring out how long a project will take, but they depend on your time estimates being accurate. If the times you assign for the tasks aren't reasonable, then those carefully built charts are nothing more than elaborate examples of GIGO (garbage in, garbage out).

One of the hardest parts of software engineering is predicting how long each task will take. One reason for this difficulty is that you rarely need to do *exactly* the same thing on multiple projects.

You may need to do something similar to something you did earlier, but the details are different enough to add some uncertainty. If a project includes a task that is *exactly* the same as one you've performed before, you can just copy the code you used before and you're done.

For example, suppose you're building an inventory application for a unicycle store, and you want to include screens that let the employees record daily timesheets. If you've build timesheet forms in a previous application, you can probably copy most or all the forms and code you wrote for the previous application and save a huge amount of time on that task.

Even if you need to make some fairly major changes, you can probably still skip a lot of the database design, form layout, and other pieces of this task that took up a lot of time when you built your first timesheet system.

Sometimes, you can take this idea even further and avoid building an entire project, either by reusing a previous project or by purchasing a commercial off-the-shelf (*COTS*) application.

COTS

CUSTOMER: I need to perform a lot of calculations, but they may change over time. Can you build something where I can enter values and equations in some sort of grid and make the program perform the calculations for me?

CONSULTANT: I could, but it would probably take a month or so and cost $20,000. What you should probably do is buy a COTS spreadsheet. It'll save you time and money and will probably be better in the long run.

CUSTOMER: Hmm. Okay. How about a program that helps me track task assignments for a long project? You know, to keep track of who's falling behind and how that will impact the final schedule?

CONSULTANT: Well, a basic program wouldn't be too hard. Maybe three weeks and $10,000 or $15,000. But you could just download some project management software. There are even free versions available if you don't need all the bells and whistles.

CUSTOMER: I see. Is there anything you *can* do for me?

CONSULTANT: I just saved you $30,000, didn't I? Ha ha.

CUSTOMER: Yes you did! Ha ha. You're fired.

The fact that you can reuse code (to some extent anyway) if you've performed the same task before means many software engineering tasks either have fairly short well-defined times, or you have little notion about how long they will take. In contrast, jobs that don't live inside cyberspace often include tasks that have well-defined durations even if they take a long time.

For example, suppose your company builds bee fences to keep elephants away from villages. You know from years of experience that it takes one person 1 day to build 50 feet of fence. If you have four employees and a customer wants a 400-foot fence, you can do some simple math to figure out how long it will take: 400 ÷ 50 ÷ 4 = 2 days.

If a new customer wants another 400-foot fence, you still know with reasonable certainty that it will take 2 days.

In contrast, suppose you want to build a bee fence design program. It will enable the user to enter some specifications and draw the fence on a map. The program will create a bill of materials, create purchase orders for the materials and either print them or transmit them to suppliers electronically, create a work schedule based on expected delivery dates, print (or transmit) an invoice for upfront costs, and generate a final invoice.

If you've never built this kind of application before, you probably don't know how long it will take to build. After you've built it, however, you won't need to build it again.

So if you assume a task's time estimate is either (1) short and well known or (2) long and highly uncertain, how can you create usable time estimates? Or are you doomed to rely on uninformed guesses? Fortunately, there are a few things you can do to minimize your risk even when you step into the great unknown.

Get Experience

One way to improve time estimates is to make the unknown known. If you can find someone who has done something similar to what you need to do, get that person to help. In smaller projects, you may be unable to pull people from other parts of your company to bring much needed experience to your team, but sometimes you can get them to help part time. They may give you time estimates that are better than random guesses, and they may give your team members some guidance about how to do the work.

Experience is even more important for long and difficult tasks. In a large project, it may be worthwhile to hire new experienced team members to tackle tricky tasks. I worked on one algorithmic project where we hired an algorithms specialist to help with maintenance. It was a good thing we did because he was the only one on the maintenance team who could understand how that part of the program worked.

Having people with previous experience can make or break a project. In fact, it's a software engineering best practice.

> **TIP** *Create a team that includes people who have done something similar before. This is particularly important for the project lead, who will give guidance to the other team members.*

Using experienced team members is the single best way to make time estimates reasonable.

Break Unknown Tasks into Simpler Pieces

Sometimes, you can break a complicated task into simple pieces that are easier to understand. In fact, that's basically what high-level and low-level designs are all about—breaking complicated tasks into simpler pieces.

For example, suppose the bee fence application needs an inventory component to track the materials your company has on hand and to order more material as needed. You may not know exactly how hard this is because you haven't done it before, but with some work (and possibly some advice from

someone who has done this sort of thing before), you can break the tasks into smaller pieces. Some of the things you'll need to do to build the inventory system include:

- Design an `Inventory` database table to store information about inventory items.
- Build a screen to let the user add and remove items from the `Inventory` table.
- Build an interface to let the program add and remove items from inventory as they are ordered and used in projects.
- Create an alert system to let someone know when inventory levels fall below a certain amount.

You may not know *exactly* how much time you'll need for each of these subtasks, but you can probably make better guesses than you could for the inventory subsystem as a whole.

> **TIP** *If you have access to people experienced with the task, have them review your breakdown before you finalize your time estimates. They may know from experience that you'll need to add a* `LeadTime` *field to the Inventory table or that creating an alert system takes a lot longer than you might expect. You'd probably discover those things as the project continued anyway, but learning this at the start will make your time estimates more accurate and may save you a lot of time and frustration trying to overcome problems that have been solved before.*

Breaking a complex task lets you convert a large unknown into several smaller pieces, which may individually be a bit less unpredictable.

Look for Similarities

Sometimes, tasks that you don't understand are at least somewhat similar to tasks that you have performed before. You may not have ordered live bees before, but you have ordered wire and fence posts, so you know at least a little about how to order supplies from distributors.

Obviously, there are some differences between bees and spools of wire, so you should expect to do some extra research before making your time estimates: Should you buy packaged bees, nucs, or established colonies? How early should you order the bees? They probably won't last as long in the warehouse as a pile of fence posts will.

As is the case in which you break large tasks into smaller ones, you should run your ideas past someone with experience if you can. They can tell you the things you missed and tell you where you may find unexpected problems.

Expect the Unexpected

Obviously, you can't predict every problem that comes along, but there are some delays that are reasonably predictable. For example, in any large project, some team members will become ill and miss some work time. They'll go on vacation and need time off for personal emergencies.

One way to handle this sort of "lost" time is to expand each task's time estimate by some amount. For example, adding 5 percent to each task's time allows for 2.6 weeks per year for vacation and sick leave.

One drawback to this approach is that people tend to use up any extra time scheduled for a task. For example, suppose a task should take 20 working days and you add an extra day to allow for lost time. If there are no problems with the task, you should finish it in 20 days and save the extra day for later in the project when you catch the plague and need to use it as sick leave. Unfortunately, most people will use up all 21 days and save nothing for later. Instead of allowing a 5-percent margin, you've basically just extended the project timeline by 5 percent.

Another approach is to add specific tasks to the project to represent lost time. If team members schedule their vacations in advance, you can include those explicitly. (Be sure not to let one team member volunteer for every "vacation time" task.)

You can also add tasks to represent sick time. Of course, you can't predict when people will get sick (with a few exceptions such as the days before and after long weekends), but you can add "sick time" tasks to the end of the schedule. Then when someone contracts a bad case of "Sunny Friday-itis," you can move the time from the "sick time" tasks to the tasks that are delayed.

Another kind of "lost time" problem occurs when your team is all geared up and raring to go but can't get anything done because of some other scheduling problem. The classic example is trying to get management approval during the holiday season. You have your project schedule worked out to the millisecond, but progress grinds to a halt because you need approval from the VP of Finance to order more highlighters and he's in Jamaica for two weeks. And don't expect your developers to be productive on August 20 if you order their computers on August 19. It's going to take some time to receive the computers, get the network running, test the e-mail system, and install *Call of Duty*.

You can avoid these kinds of problems by carefully planning for approvals, order lead times, and setup. In fact, while you're at it, you may as well make them tasks and put them in the schedule.

Track Progress

Even if you have previous experience with a type of task, break the task into smaller pieces, plan for "lost time," and allow a buffer for unexpected problems, sometimes things just take longer than you expect. It's extremely important to keep track of tasks as they progress and take action if one is not going according to plan.

For example, suppose a task was scheduled to take 20 days. After five days, you ask the developer assigned to that task how much is done and he says he's 25-percent complete and has 15 days of work remaining. In another week, the developer says he's 50-percent done and has 10 days of work remaining. So far so good, but after another week, the developer says he's 60-percent done (when he should be 75-percent finished).

Initially, it seemed like he was making good progress but as the task's completion date draws near, the developer realizes how much work is left to do. This is normal and not necessarily a cause for panic, but it does require attention. You need to dig deeper and find out if the developer can actually finish the task on time or if you need to adjust the schedule.

In reality the developer is just making the best guesses he can. If he hasn't performed a task like this one before, those guesses may not be perfect. In this example, the developer's first two estimates were probably off, so he had actually completed only 20 percent of the task after one week and 40 percent after two weeks. If that's the case, he probably needs another two weeks to finish the task instead of the one week that's scheduled. (Of course, that assumes the third estimate of 60 percent is correct, and it may not be.)

Many developers are naturally optimistic and assume they can make up lost time, but they're often wrong. Or they can make up the time but only by working ridiculously long hours. Working extra hours once in a while is okay, but developers who work extra hours too often eventually burn out. (Keeping a sustainable pace is one of the core principles behind agile development, which is discussed in Chapter 14, "RAD.")

At this point, you may want to add the extra week to the task and see what happens to the rest of the project's schedule.

80-PERCENT RIGHT, 50 PERCENT OF THE TIME

The whole schedule depends on time estimates that are uncertain at best. You make estimates for each task and, during development, the developers make estimates about how much work they have finished and how much work they have left to do.

It's hard to make estimates more accurate (that requires experience), but it's easy to make estimates worse. All you need to do is to yell at the developers, draw lines in the sand, talk about red lines and points of no return, brag about how you *don't* miss deadlines, and generally throw management-speak at the developers. If you make it clear that developers have to stay on track *at all costs*, their estimates will show that they *are* on track... right up to the point at which they miss their deadlines.

A much better approach is to encourage developers to give you estimates that are as truthful and accurate as possible. Over time you'll figure out who can make good estimates and who's always off by 15 percent. That improves your ability to plan and that greatly increases your chances of success.

If the developer can get the task back on schedule, that's great, but you should pay extra attention to that task to see if the latest 60 percent estimate is correct or if the task is really in more trouble than the developer thinks. The biggest mistake you can make is to ignore the problem and hope you can make up the time later. Unless you have a reason to believe you can catch up, you need to assume you'll fall farther behind. Working on a task that continues to slip week after week feels like you're trying to bail out a sinking lifeboat with a colander.

Possibly the second biggest mistake you can make is to pile extra developers on the task and assume they can reduce the time needed to finish it. As Fred Brooks said in his famous book *The Mythical Man-Month*, "adding manpower to a late software project makes it later." Adding someone with appropriate expertise to a task can sometimes help, but it takes time for new people to get up to speed on any task, so you shouldn't just throw more bodies at a task and hope that'll help. This feels like someone's thrown more people into your sinking lifeboat. You may like the company, but unless they have a working pump, their weight will just make you sink faster.

Risk Management

The preceding section talks about task-tracking and how you can react if a task starts to slip. Often you need to add extra time to the schedule and keep a close watch on that task. Sometimes, you can

add extra people to the task; although, bringing them up to speed may actually slow the task down if they don't bring particularly useful expertise to the job.

Risk management is more proactive. Instead of responding to problems after they occur, risk management identifies possible risks, determines their potential impacts, and studies possible work-arounds ahead of time.

For each task, you should determine:

- **Likelihood**—Do you know more or less how to perform this task? Or is this something you've never done before so it might hold unknown problems?
- **Severity**—Can the users live without this feature if the task proves difficult? Can you cancel this feature or push it into a future release?
- **Consequences**—Will problems with this task affect other tasks? If this task fails, will that cause other tasks to fail or make other tasks unnecessary?
- **Work-arounds**—Are there work-arounds? What other approaches could you take to solve this problem? For each work-around consider:
  - **Difficulty**—How hard will it be to implement this work-around? How long will it take? What are the chances that this work-around will work?
  - **Impact**—What affects do the work-arounds have on the project's usability? Is this going to make a lot of extra work for the users?
  - **Pros**—What are the work-arounds' advantages?
  - **Cons**—What are the work-arounds' disadvantages?

You can use your analysis to study how different kinds of problems will affect the schedule. For example, suppose a task is harder than you originally planned. You've used 5 of the 10 days allocated to the task but you haven't really made any progress. If that task's risk analysis includes a sure-fire work-around that provides an acceptable alternative and you're quite sure would take eight days to implement, you might want to switch to the work-around and take the 3-day schedule slip rather than following the original approach with its unknown duration.

EXAMPLE Example Risky Reorders

In this example, let's perform risk analysis on a reordering feature for the bee fence design application.

Suppose you want the bee fence application to automatically place orders for staples, envelopes, fence posts, and other supplies whenever inventory runs low. Unfortunately, you've never written code to do that before. This is a possible point of risk because you don't know how to do it, and it may be much harder to do than you think it is.

In this case, you might use the following notes to describe this task's risk:

- **Task**—Reorder inventory. (Reorder when inventory is low.)
- **Likelihood**—Medium. (We don't really know whether this will be a problem, but the likelihood is definitely not low.)

- **Severity**—High. (We need some way to reorder supplies or we'll be out of business!)
- **Consequences**—None. (Except, obviously, for the company going bankrupt and all the employees ending up on the street begging for spare change. By "None" I mean there are no other tasks that depend on this one working as originally planned.)
- **Work-around 1**—Send an e-mail to an administrator who then places the order manually.
  - **Difficulty**—Easy. I've done this before and it's not too hard. Estimated time: 3 days.
  - **Impact**—This change would make about 1 hour more of work for the administrator per month.
  - **Pros**—Simple. Keeps a person in the loop. (A programming bug can't make the administrator accidentally order 1 million fence posts or 12,000 miles of wire.)
  - **Cons**—Not automatic. The administrator needs to follow through. Will need some sort of backup when the administrator is out of the office.
- **Work-around 2**—Send a text message to an administrator who then places the order manually.
  - **Difficulty**—Easy. (Similar to Work-around 1).
  - **Impact**—Similar to Work-around 1.
  - **Pros**—Similar to Work-around 1. The administrator receives notification even if not at work. Could be added in addition to Work-around 1 for little extra work.
  - **Cons**—The administrator might forget to place the order, particularly if he receives the message while away from work.

After you've performed the risk analysis on the reorder inventory task, you can use it in your project planning. For example, you could allow 5 days to do this task. If you haven't made good progress in the first 2 days, you can drop back to Work-around 1 and push automatic reordering to the second release.

SUMMARY

As much as some programmers might like to deny it, management is an important part of software engineering. Executive management is essential for the project to succeed. Project management is critical for scheduling and tracking tasks to make sure the project moves toward completion instead of into a morass of side issues and never-ending tasks.

PERT charts, critical path methods, and Gantt charts can help a project manager keep things on track, but they won't do any good unless you have reasonable time estimates. Techniques such as using experienced team members, breaking large tasks into smaller pieces, and allowing for unexpected lost time can make time estimates more accurate.

Even if you use every conceivable time estimation trick, unexpected surprises can throw a monkey wrench into the works. Risk management lets you handle those sorts of unpredictable disasters

quickly and efficiently. If a task looks like it will be impossible or greatly delayed, you can switch to a work-around to stay on track and still produce something usable.

This chapter and the two previous ones provide background that you need before you move on to actually starting a new software project. The next chapter describes the first step in building a new application: requirements gathering.

EXERCISES

Okay, I admit the zombie apocalypse bunker project isn't software-related...*unless* you decide to write a 3-D computer game based on that concept! Sort of *World of Warcraft* meets *World War Z*. You could call it *World of Z-Craft*.

Table 3-2 summarizes some of the classes and modules you might need (and their unreasonably optimistic expected times) to develop players and zombies for the game. (The program would also need lots of other pieces not listed here to handle other parts of the game.)

TABLE 3-2: Classes and Modules for World of Z-Craft

TASK	TIME (DAYS)	PREDECESSORS
A. Robotic control module	5	—
B. Texture library	5	C
C. Texture editor	4	—
D. Character editor	6	A, G, I
E. Character animator	7	D
F. Artificial intelligence (for zombies)	7	—
G. Rendering engine	6	—
H. Humanoid base classes	3	—
I. Character classes	3	H
J. Zombie classes	3	H
K. Test environment	5	L
L. Test environment editor	6	C, G
M. Character library	9	B, E, I
N. Zombie library	15	B, J, O
O. Zombie editor	5	A, G, J
P. Zombie animator	6	O
Q. Character testing	4	K, M
R. Zombie testing	4	K, N

1. Draw a PERT chart for these tasks. Include the tasks' letters, predecessors, and expected times.

2. Use critical path methods to find the total expected time from the project's start for each task's completion. Find the critical path. What are the tasks on the critical path? What is the total expected duration of the project in working days?

3. How long is the second-shortest path in the PERT network you built for Exercise 2? What tasks lie along a second-longest path? By how much could the tasks on the path slip before impacting the project's total time?

4. Build a Gantt chart for the network you drew in Exercise 3. Start on Wednesday, January 1, 2020, and don't work on weekends or the following holidays:

HOLIDAY	DATE
New Year's Day	January 1
Martin Luther King Day	January 20
President's Day	February 17

(These are U.S. holidays. If you live somewhere else, feel free to use your own holidays.)

On what date do you expect the project to be finished?

5. Download a trial version of the project management tool of your choice and use it to enter the zombie apocalypse tasks. Does it agree with the end date you found in Exercise 4? (The Internet is crawling with useful project management tools, so you have lots of choices. The solution shown in Appendix A, "Solutions to Exercises," uses OpenProj. It's simple and you can download it for free at `OpenProj.org`.) What are the advantages and disadvantages of using the tool you selected over building a Gantt chart manually?

6. In addition to losing time from vacation and sick leave, projects can suffer from problems that just strike out of nowhere. Sort of a bad version of *deus ex machina*. For example, senior management could decide to switch your target platform from Windows desktop PCs to the latest smartwatch technology. Or a strike in the Far East could delay the shipment of your new servers. Or one of your developers might move to Iceland. How can you handle these sorts of completely unpredictable problems?

7. What techniques can you use to make accurate time estimates?

8. What are the two biggest mistakes you can make while tracking tasks?

▸ WHAT YOU LEARNED IN THIS CHAPTER

- Executive support is critical for project success.
- A project manager schedules and tracks tasks, and keeps developers moving forward.
- PERT charts show precedence relationships among tasks.
- While building a PERT chart, if you can't find a task with no unsatisfied predecessors, the tasks contain a precedence loop and no schedule is possible.
- Critical path methods show the longest paths through a PERT network. If a task on one of those paths is delayed, the project's final completion is delayed.
- Gantt charts show task durations, start time, and end times.
- You can improve time estimates by using experience, breaking complex tasks into smaller tasks, and looking for similarities to previous tasks.
- You should plan for delays such as illness, vacation, and unexpected problems.
- Risk management lets you plan for problems so that you can react quickly when they occur.

4

Requirement Gathering

If you don't know where you are going, you'll end up someplace else.

—Yogi Berra

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Why requirements are important
- The characteristics of good requirements
- The MOSCOW method for prioritizing requirements
- Audience-oriented, FURPS, and FURPS+ methods for categorizing requirements
- Methods for gathering customer goals and turning them into requirements
- Brainstorming techniques
- Methods for recording requirements such as formal specifications, user stories, and prototypes

It's tempting to say that requirement gathering is the most important part of a software project. After all, if you get the requirements wrong, the resulting application won't solve the users' problems. You'll be like a tourist in Boston with a broken GPS. You may get somewhere interesting, but you probably won't get where you want to go.

Even though requirements are important for setting a project's direction, a project can fail at any other stage, too. If you build a flawed design, write bad code, fail to test properly, or even provide incorrect training materials, the project can still fail. If any one of the links in the development chain fails, the project will fail.

Let's just say that requirement gathering is the first link in the chain, so it's the first place where you can screw things up badly. Requirements *do* set the stage for everything that follows, so while you can argue over whether this is the most important step, it's definitely *an* important step.

This chapter explains what requirement gathering is and lists some typical requirements that are useful in many projects. It also describes some techniques you can use to gather requirements effectively.

REQUIREMENTS DEFINED

Requirements are the features that your application must provide. At the beginning of the project, you gather requirements from the customers to figure out what you need to build. Throughout development, you use the requirements to guide development and ensure that you're heading in the right direction. At the end of the project, you use the requirements to verify that the finished application actually does what it's supposed to do.

Depending on the project's scope and complexity, you might need only a few requirements, or you might need hundreds of pages of requirements. The number and type of requirements can also depend on the level of formality the customers want.

For example, if you're working on a casual in-house project, your boss may be satisfied with a few blanket requirements such as "find ways to improve order processing" or "write a tool to send spam to customers." As long as you create something vaguely useful, your project will probably be viewed as a success. (If not, you'll find out at your annual review.) As you'll see shortly, these sorts of vague requirements have some problems.

Large projects with higher stakes typically have far more requirements that are spelled out much more formally and in great detail. For example, if you're building an autopilot system for 747s or you're writing software to control pacemakers, your requirements must be unambiguous. You can't wait until the final weeks of testing to start thinking about whether "easy installation" means patients should change their own pacemaker parameters from a cell phone.

The following sections describe some of the properties that requirements should have to be useful.

Clear

Good requirements are clear, concise, and easy to understand. That means they can't be pumped full of management-speak, florid prose, and confusing jargon.

It is okay to use technical terms and abbreviations if they are defined somewhere or they are common knowledge in the project's domain. For example, when I worked at a phone company research lab, we often used terms like POTS (plain old telephone service), PBX (public branch exchange), NPA (numbering plan area, known to nontelephone people as an area code), and ISDN (integrated services digital network, or as some of us used to call it, "I still don't know"). The customers and development team members all knew those terms, so they were safe to use in the requirements.

To be clear, requirements cannot be vague or ill-defined. Each requirement must state in concrete, no-nonsense terms exactly what it requires.

For example, suppose you're working on a program to schedule appointments for utility repair people. (Those appointments that typically say, "We'll be there sometime between 6:00 a.m. and midnight during the next 2 weeks.") A requirement such as, "Improve appointment scheduling," is too vague to be useful. Does this mean you should tighten the appointment windows even if it means missing more appointments? Does it mean repair people should leave and make a new appointment if they can't finish a job within 1 hour? Or does it mean something crazy like letting customers tell you what times they can actually be home and then fitting appointments to those times?

A better requirement would be, "Reduce appointment start windows to no more than 2 hours while meeting 90 percent of the scheduled appointments."

Unambiguous

In addition to being clear and concrete, a requirement must be unambiguous. If the requirement is worded so that you can't tell what it requires, then you can't build a system to satisfy it. Although this may seem like an obvious feature of any good requirement, it's sometimes harder to guarantee than you might think.

For example, suppose you're building a street map application for inline skaters, and you have a requirement that says the program will, "Find the best route from a start location to a destination location." This can't be all that hard. After all, Google Maps, Yahoo Maps, MapQuest, Bing Maps, and other sites all do something like this.

But how do you define the "best" route? The shortest route? The route that uses only physically separated bike paths so that the user doesn't have to skate in the street? Or maybe the route that passes the most Starbucks locations?

Even if you decide the "best" route means the shortest one, what does that mean? The route that's the shortest in distance? Or the shortest in time? What if the route of least distance goes up a steep hill or down a set of stairs and that increases its time? (In this example, you might change the requirements to let the users decide how to pick the "best" route at run time.)

As you write requirements, do your best to make them unambiguous. Read them carefully to make sure you can't think of any way to interpret them other than the way you intend.

Then run them past some other people (particularly customers and end user representatives) to see if they agree with you.

A TIMELY JOKE

CUSTOMER: I need you to write a program to find customers that haven't paid their bills within 5 seconds.

DEVELOPER: Harsh! Most companies give their customers 30 days to pay their bills.

Consistent

A project's requirements must be consistent with each other. That means not only that they cannot contradict each other, but that they also don't provide so many constraints that the problem is unsolvable. Each requirement must also be self-consistent. (In other words, it must be possible to achieve.)

Consider again the earlier example of utility repair appointments. You might like to include the following two requirements:

- Reduce appointment start windows to no more than 2 hours.
- Meet 90 percent of the scheduled appointments.

It may be that you cannot satisfy these two requirements at the same time. (At least using only software. You might do it if you hire more repair people.)

In a complex project, it's not always obvious if a set of requirements is mutually consistent. Sometimes, any pair of requirements is satisfiable but larger combinations of requirements are not.

A common software engineering expression is, "Fast, good, cheap. Pick two." The idea is you can trade development speed, development quality, and cost, but you can't win in all three dimensions. Only three possible combinations work:

- Build something quickly with high quality and high cost.
- Build something quickly and inexpensively but with low quality.
- Build with high quality and low cost but over a long time.

Try to keep new requirements consistent with existing requirements. Or rewrite older requirements as necessary. When you finish gathering all the requirements, go through them again and look for inconsistencies.

Prioritized

When you start working on the project's schedule, it's likely you'll need to cut a few nice-to-haves from the design. You might like to include every feature but don't have the time or budget, so something's got to go.

At this point, you need to prioritize the requirements. If you've assigned costs (usually in terms of time to implement) and priorities to the requirements, then you can defer the high-cost, low-priority requirements until a later release.

Customers sometimes have trouble deciding which requirements they can live without. They'll argue, complain, and generally act like you're asking which of their children they want to feed to the dingoes. Unfortunately, unless they can come up with a big enough budget and timescale, they're going to need to make some sort of decision.

The exception occurs when you work on life-critical applications such as nuclear reactor cooling, air traffic control, and space shuttle flight software. In those types of applications, the customer may have a lot of "must have" requirements that you can't remove without compromising the applications' safety. You may remove cosmetic requirements like a space shuttle's automatic

turn-signal cancellation feature, but you're probably going to need to keep the fuel monitor and flight path calculator.

THE MOSCOW METHOD

MOSCOW is an acronym to help you remember a common system for prioritizing application features. The consonants in MOSCOW stand for the following:

M—Must. These are ***required*** features that must be included. They are necessary for the project to be considered a success.

S—Should. These are ***important*** features that should be included if possible. If there's a work-around and there's no room in the release 1 schedule, these may be deferred until release 2.

C—Could. These are ***desirable*** features that can be omitted if they won't fit in the schedule. They can be pushed back into release 2, but they're not as important as the "should" features, so they may not make it into release 2, either.

W—Won't. These are ***completely optional*** features that the customers have agreed will not be included in the current release. They may be included in a future release if time permits. (Or they may just be included in the requirements list to make a particularly loud and politically connected customer happy, and you have no intention of ever including these features.)

Let's face it. If a feature isn't a "must" or "should," then its chances of ever being implemented are slim. After this release has been used for a while, you'll probably receive tons of bug reports, requests for changes, and pleas for new features, so in the next release you still won't have time for the "could" and "won't" features.

(Unless you're one of these big software companies, who shall remain nameless, that thinks it needs to push a new version of its products out every 2 years to make customers buy something. Sometimes those products reach deep into the "could" and "won't" categories, and perhaps even the "why?" and "you must be joking!" categories.)

EXAMPLE ClassyDraw

For an example of using the MOSCOW method, consider a fictional drawing program named ClassyDraw. It's somewhat similar to MS Paint and allows you to draw line segments, ellipses, polygons, text, and other shapes. The big difference is that ClassyDraw represents each shape you draw as an object that you can later select, move, resize, modify, and delete.

Here's an initial requirement list:

1. Draw: line segments, sequences of line segments, splines, polygons, ellipses, circles, rectangles, rounded rectangles, stars, images, and other shapes.
2. Save and load files.

3. Protect the current drawing. For example, if the user tries to close the program while there are unsaved changes, prompt the user.
4. Let the user specify the line style and colors used to draw shapes.
5. Let the user specify the fill style and colors used to draw shapes.
6. Click to select an object.
7. Click and drag to select multiple objects.
8. Click or click and drag with the Shift key down to add objects to the current selection.
9. Click or click and drag with the Ctrl key down to toggle objects in and out of the current selection.
10. Click and drag the selected objects to move them.
11. Edit the selected objects' line and fill styles.
12. Delete the selected objects.
13. Select colors from a palette.
14. Place custom colors in a custom palette.
15. Support transparency.
16. Copy and paste the entire drawing, a rectangular selection, or an irregular selection as a bitmapped image.
17. Copy, cut, and paste the currently selected objects.
18. Allow the user to write scripts to add shapes to a drawing.
19. Let the user rearrange the palettes and toolbars.
20. Auto-save the current drawing periodically. If the program crashes, allow the user to reload the most recently saved version.
21. Auto-save the current drawing every time a change is made. If the program crashes, allow the user to reload the most recently saved version.
22. Provide online help.
23. Provide online tutorials.

Now you can use the MOSCOW method to prioritize these requirements.

Must. To identify the "must" requirements, examine each requirement and ask yourself: Could that requirement be omitted? Would the program be usable without that feature? Will users give the product 1-star reviews and say they wish they could give 0 stars? That the product would be overpriced if it were freeware?

The ClassyDraw application ***must*** be able to save and load files (2). You could build early test versions that couldn't, but it would be unacceptable to users.

Similarly the program must ensure the safety of the current drawing (3). The users would never forgive the program if it discarded a complicated drawing without any warning.

The program wouldn't be useful if it didn't draw, so the program must draw at least a few shapes (1). For starters, it could draw line segments, rectangles, and ellipses. You could add more shapes in later releases.

The program should probably allow the user to click objects to select them. Otherwise, the user may as well use MS Paint, so requirement 6 is a must. Of course, there's little point in selecting an object if you can't do anything with it, so the program must let the user at least move (10) and delete (12) the selected objects.

The "must" requirements include 1 (partial), 2, 3, 6, 10, and 12.

Should. To identify the "should" requirements, examine each of the remaining requirements and ask yourself, "Does that feature significantly enhance the product? If it were omitted, would users be constantly asking why it wasn't included? Is the feature common in other, similar applications? Will users give the product 2-star and 3-star reviews?"

Several requirements that are fairly standard for drawing applications didn't make the cut for the "must" category. (You could say they didn't pass muster.)

Most (if not all) of the other shapes in requirement 1 should be included in this group. When drawing new shapes, the user should also indicate the line and fill styles the new shapes should have (4, 5). That will require specifying colors, at least from a palette (13).

The click-and-drag selection technique (7) should be included, as should the ability to hold down the Shift or Ctrl key while making selections (8, 9).

Any decent application should have help (22) and documentation (23), so those should also be included.

The "should" requirements include 1 (remaining), 4, 5, 7, 8, 9, 13, 22, and 23.

Could. To identify the "could" requirements, examine each of the remaining requirements and ask yourself, "Would that requirement be useful to the users? Is it something special that other similar applications don't have? Will this help bump reviews up to 4 or 5 stars? Is this a feature that we should include at some point, just not in the first release?"

Another way to approach this category is to ask: Which features will we need in the long term? Which of the remaining features shouldn't be dumped in the trash heap labeled "won't"?

Most of the remaining requirements should probably not go in the "won't" pile. If they were that bad , they probably wouldn't have made it into the requirements list in the first place.

The "could" category should definitely include the ability to edit selected objects (11). This is another of the main reasons for allowing the user to select objects.

Support for custom colors (14) and transparency (15) would also be nice, if time permits. Cut, copy, and paste for images (16) and selected objects (17) would be useful, so they should be included.

The "could" requirements include 11, 14, 15, 16, and 17.

Won't. To identify the "won't" requirements, examine the remaining requirements and ask yourself, "Is this unnecessary, confusing, or just plain stupid? Will it be used only rarely? Does it add nothing useful to the application?" If you can't answer "yes" to those questions for a particular requirement, then you should think about moving that requirement into one of the other categories.

For this application, allowing users to write scripts (18) would be cool but probably rarely used. Letting the user rearrange palettes and toolbars (19) would be a nice touch, but isn't important.

Auto-saving (20, 21) is also a nice touch, but probably unnecessary. We can look at user requests and conduct surveys to see if this feature would be worth adding to a future release.

The "won't" requirements include 18, 19, 20, and 21.

After you've assigned each requirement to a category, go back through them and make sure you're happy with their assignments. If a requirement in the "could" category seems more important than one in the "should" category, switch them.

Also make sure every requirement is in some category and that every category contains some requirement. If every requirement is in the "must" category, then you may need to rethink your priorities (or your customer's priorities), or be sure you'll have enough time to get everything done.

Verifiable

Requirements must be verifiable. If you can't verify a requirement, how do you know whether you've met it?

Being verifiable means the requirements must be limited and precisely defined. They can't be open-ended statements such as, "Process more work orders per hour than are currently being processed." How many work orders is "more?" Technically, processing one more work order per hour is "more," but that probably won't satisfy your customer. What about 100? Or 1,000?

A better requirement would say, "Process at least 100 work orders per hour." It should be relatively easy to determine whether your program meets this requirement.

Even with this improved requirement, verification might be tricky because it relies on some assumptions that it doesn't define. For example, the requirement probably assumes you're processing work orders in the middle of a typical workday, not during a big clearance event, during peak ordering hours, or during a power outage.

An even better requirement might be, "Process at least 100 work orders per hour on average during a typical work day." You may want to refine the requirement a bit to try to say what a "typical work day" is, but this version should be good enough for most reasonable customers.

Words to Avoid

Some words are ambiguous or subjective, and adding them to a requirement can make the whole thing fuzzy and imprecise. The following list gives examples of words that may make requirements less exact.

- **Comparatives**—Words like faster, better, more, and shinier. How much faster? Define "better." How much more? These need to be quantified.
- **Imprecise adjectives**—Words like fast, robust, user-friendly, efficient, flexible, and glorious. These are just other forms of the comparatives. They look great in management reports, business cases, and marketing material, but they're too imprecise to use in requirements.

- **Vague commands**—Words like minimize, maximize, improve, and optimize. Unless you use these in a technical algorithmic sense (for example, if you optimize flow through a network), these are just fancy ways to say, "Do your best." Even in an algorithmic sense, these sorts of words are often applied to hard problems where exact solutions may not exist. In any case, you need to make the goals more concrete. Provide some numbers or other criteria you can use to determine whether a requirement has been met.

REQUIREMENT CATEGORIES

In general, requirements tell what an application is supposed to do. Good requirements share certain characteristics (they're clear, unambiguous, consistent, prioritized, and verifiable), but there are several kinds of requirements that are aimed at different audiences or that focus on different aspects of the application. For example, business requirements focus on a project's high-level objectives and functional requirements give the developers more detailed lists of goals to accomplish.

Assigning categories to your requirements isn't the point here. (Although there are two kinds of people in the world: those who like to group things into categories and those who don't. If you're one of the former, then you may need to do this for your own peace of mind.) The real point here is that you can use the categories as a checklist to make sure you've created requirements for the most important parts of the project. For example, if you look through the requirements and the reliability category is empty, you might consider adding some new requirements.

You can categorize requirements in several ways. The following sections describe four ways to categorize requirements.

Audience-Oriented Requirements

These categories focus on different audiences and the different points of view that each audience has. They use a somewhat business-oriented perspective to classify requirements according to the people who care the most about them.

For example, the corporate vice president of Plausible Deniability probably doesn't care too much about which button a call center clerk needs to press to launch a customer into a never-ending call tree as long as it works. In contrast, the clerk needs to know which button to press.

The following sections describe some of the more common business-oriented categories.

Business Requirements

Business requirements lay out the project's high-level goals. They explain what the customer hopes to achieve with the project.

Notice the word "hopes." Customers sometimes try to include all their hopes and dreams in the business requirements in addition to verifiable objectives. For example, they might say the project will "Increase profits by 25 percent" or "Increase demand and gain 10,000 new customers." Although those goals have numbers in them, they're probably outside the scope of what you can achieve through software engineering alone. They're more like marketing targets than project requirements. You can craft the best application ever put together, but someone still needs to use it properly to realize the new profits and customers.

Sometimes, those vague goals are unavoidable in business requirements, but if possible you should try to push them into the business case. The business case is a more marketing-style document that attempts to justify the project. Those often include graphs and charts showing projected costs, demand, sales figures, and other values that aren't known exactly in advance.

To think of this another way, I have no qualms about promising to write a system that can pull up a customer's records in less than 3 seconds or find the closest donut shop that's open at 2 a.m. (if you give me the data). But I wouldn't want to promise to improve morale in the Customer Complaints department by 15 percent. (What would that even mean?)

User Requirements

User requirements (which are also called *stakeholder requirements* by managers who like to use the word "stakeholder"), describe how the project will be used by the eventual end users. They often include things like sketches of forms, scripts that show the steps users will perform to accomplish specific tasks, use cases, and prototypes. (The sections "Use Cases" and "Prototypes" later in this chapter say more about the last two.)

Sometimes these requirements are very detailed, spelling out exactly what an application must do under different circumstances. Other times they specify *what* the user needs to accomplish but not necessarily *how* the application must accomplish it.

EXAMPLE Overly Specific Selections

In this example, you see how you can turn an overly specific requirement into one that's flexible without making it vague.

Suppose you're building a phone application that lets customers place orders at a sandwich and bagel shop called The Loxsmith. The program should let customers select the toppings they want on their bagels. They include lox (naturally), butter, cream cheese, gummy bears, and so on. Here's one way you could word this requirement:

> *The toppings form will display a list of toppings. The user can check boxes next to the toppings to add them to the bagel.*

That's a fine requirement. Clear, concise, verifiable. Everything you could want in a requirement. Unfortunately, it's also unnecessarily specific. It forces the designers and developers to use a specific technique to achieve the higher-level goal of letting the customer select toppings.

During testing, you might discover that The Loxsmith provides more than 200 toppings. In that case, the program won't be able to display a list of every topping at the same time. The user will need to scroll through the list, and that will make it hard for the customer to see what toppings are selected.

Here's a different version of the same requirement that doesn't restrict the developers as much.

> *The toppings form will allow the user to select the toppings put on the bagel.*

The difference is small but important. With this version, the developers can explore different methods for selecting toppings. If you have user-interface specialists on your team, they may create a variety of

possible solutions. For example, customers might drag and drop selections from a big scrollable list on the left onto a shorter list of selected items on the right. Then they could always see what toppings were selected. You might even display a cartoon picture of a bagel holding the user's four dozen selected toppings piled up like the Leaning Tower of Pisa.

Vague requirements are bad, but flexible requirements let you explore different options before you start writing code. To keep requirements as flexible as possible, try to make the requirements spell out the project's *needs* without mandating a particular approach.

Functional Requirements

Functional requirements are detailed statements of the project's desired capabilities. They're similar to the user requirements but they may also include things that the users won't see directly. For example, they might describe reports that the application produces, interfaces to other applications, and workflows that route orders from one user to another during processing.

These are things the application should do.

Note that some requirements could fall into multiple categories. For example, you could consider most user requirements to be functional requirements. They not only describe a task that will be performed by the user, but they also describe something that the application will do.

Nonfunctional Requirements

Nonfunctional requirements are statements about the quality of the application's behavior or constraints on how it produces a desired result. They specify things such as the application's performance, reliability, and security characteristics.

For example, a functional requirement would be, "Allow users to reserve a hovercraft online." A nonfunctional requirement would be, "The application must support 20 users simultaneously making reservations at any hour of the day."

Implementation Requirements

Implementation requirements are temporary features that are needed to transition to using the new system but that will be later discarded. For example, suppose you're designing an invoice-tracking system to replace an existing system. After you finish testing the system and are ready to use it full time, you need a method to copy any pending invoices from the old database into the new one. That method is an implementation requirement.

The tasks described in implementation requirements don't always involve programming. For example, you could hire a bunch of teenagers on summer break to retype the old invoices into the new system. (Although you'll probably get a quicker and more consistent result if you write a program to convert the data into the new format. The program won't get bored and stop coming to work when the next release of *Grand Theft Auto* comes out.)

Other implementation requirements include hiring new staff, buying new hardware, preparing training materials, and actually training the users to use the new system.

FURPS

FURPS is an acronym for this system's requirement categories: functionality, usability, reliability, performance, and scalability. It was developed by Hewlett-Packard (and later extended by adding a + at the end to get FURPS+).

The following list summarizes the FURPS categories:

- **Functionality**—What the application should do. These requirements describe the system's general features including what it does, interfaces with other systems, security, and so forth.
- **Usability**—What the program should look like. These requirements describe user-oriented features such as the application's general appearance, ease of use, navigation methods, and responsiveness.
- **Reliability**—How reliable the system should be. These requirements indicate such things as when the system should be available (12 hours per day from 7:00 a.m to 8:00 p.m.), how often it can fail (3 times per year for no more than 1 hour each time), and how accurate the system is (80 percent of the service calls must start within their predicted delivery windows).
- **Performance**—How efficient the system should be. These requirements describe such things as the application's speed, memory usage, disk usage, and database capacity.
- **Supportability**—How easy it is to support the application. These requirements include such things as how easy it will be to maintain the application, how easy it is to test the code, and how flexible the application is. (For example, the application might let users set parameters to determine how it behaves.)

FURPS+

FURPS was extended into FURPS+ to add a few requirements categories that software engineers thought were missing. The following list summarizes the new categories:

- **Design constraints**—These are constraints on the design that are driven by other factors such as the hardware platform, software platform, network characteristics, or database. For example, suppose you're building a financial application and you want an extremely reliable backup system. In that case, you might require the project to use a shadowed or mirrored database that stores every transaction off-site in case the main database crashes.
- **Implementation requirements**—These are constraints on the way the software is built. For example, you might require developers to meet the Capability Maturity Model Integration (CMMI) or ISO 9000 standards. (For more information on those, see `www.cmmifaq.info` and `www.iso.org/iso/iso_9000` respectively.)
- **Interface requirements**—These are constraints on the system's interfaces with other systems. They tell what other systems will exchange data with the one you're building. They describe things like the kinds of interactions that will take place, when they will occur, and the format of the data that will be exchanged.
- **Physical requirements**—These are constraints on the hardware and physical devices that the system will use. For example, they might require a minimum amount of processing power, a maximum amount of electrical power, easy portability (such as a tablet or smartphone), touch screens, or environmental features (must work in boiling acid).

EXAMPLE FURPS+ Checklist

In this example, we'll use FURPS+ to see if any requirements are missing for the The Loxsmith ordering application. Consider the following abbreviated list of requirements. The program should allow the user to:

Start an order that might include multiple items.

Select bagel type.

Select toppings.

Select sandwich bread.

Select sandwich toppings.

Select drinks.

Select pickup time.

Pay or decide to pay at pickup.

I've left out a lot of details from this list such as the specific bagel, bread, and topping types that are available, but at first glance, this seems like a reasonable set of requirements. It describes what the application should do but doesn't impose unnecessary constraints on how the developers should build it. It's a bit more vague than I would like (how do you *verify* that the user can select toppings?), but you can flesh that out. (In fact, I'll talk a bit about ways you can do that later in this chapter, particularly when I talk about use cases in the section "Use Cases.")

For this example, assume the requirements are spelled out in specific (but flexible) detail. Then use FURPS+ to see if there's anything important missing from this list. Spend a few minutes to decide in which FURPS+ category each of the requirements belongs.

Although the initial requirements all seem reasonable, they're all functionality requirements. They tell what the application should do but don't give much information about usability, reliability, performance, and other requirements that should belong to the other FURPS+ categories.

You might think that a requirements list containing only functionality requirements would be an unusual situation. However, left to their own devices, many programmers come up with exactly this sort of list. They focus on the work they are going to do and how it will look to the users. That's a good place to start the design, but in the background they're making a huge number of assumptions about things they take for granted.

For example, suppose you're a developer who writes Java applications running on Android tablets. In that case, you may think the previous list of requirements is just fine. Your version of the Eclipse Java development environment is up to date, you've installed the Android Software Development Kit (SDK), and you have "Eye of the Tiger" blasting on your headphones. You're ready to start cranking out code.

Unfortunately you're also making a ton of assumptions that may or may not sit well with the customer. In this example, you're assuming the application will be written in Java to run on Android tablets. What if the customer wants the application to run on an iPhone, Windows Phone, mobile-oriented web page, Google Glass, or some sort of smart wearable ankle bangle device? Or maybe all of the above?

Sometimes, you may not want any requirements in a particular category, but the fact that the preceding list contains ***only*** functionality requirements is a strong hint that we're doing something wrong. You

should at least think about every category and either (1) come up with some new requirements that belong there, or (2) write down why you don't think you need any requirements for that category.

So now look at the FURPS+ requirement categories:

Functionality—(What the program should do.) The initial list of requirements covers this category.

Usability—(What the program should look like.) You could add some requirements indicating how the user navigates from starting an order to picking sandwich and bagel ingredients. You could also provide details about login (should we create customer accounts?) and the checkout method. You should also specify that each form will display The Loxsmith logo.

Reliability—(How reliable the system should be.) Should the application be available only while The Loxsmith is open? Or should customers be able to pre-order a morning jalapeno popper bagel and double kopi luwak to pick up on the way in to work?

Performance—(How efficient the system should be.) How quickly should the application respond to customers (assuming they have a fast Internet connection)?

Supportability—(How easy should the system be to support?) The requirements should indicate that The Loxsmith employees can edit the information about the types of breads, bagels, toppings, and other items that are available. You might also want to add automated testing requirements, information about help available to customers, and any plans for future versions of the project.

Design—(Design constraints.) Here's where you would specify the target hardware and software platforms. For example, you might want the program to run on iPhones (code written with Xcode) and Windows Phones (code written in C#).

Implementation—(Constraints on the way the software is built.) You can specify software standards. For example, you might require pair programming or agile methods. (Those are described in Chapter 14, "RAD.")

Interface—(Interfaces with other systems.) Perhaps you want the application to call web services that use Simple Object Access Protocol (SOAP) to let other programs place sandwich orders. (Although it's not clear how many other companies will want an automated ordering interface to The Loxsmith, so perhaps this category will be intentionally left blank.)

Physical—(Hardware requirements.) For this application, the customers provide their own hardware (such as phones and tablets) so you don't need to specify those. You might want to specify the server hardware. Or you might want to lease space on an Internet service provider so that you don't need to buy your own hardware. (You should probably still study the available options so that you know how powerful they are and how much they cost.)

Using requirements categories as a checklist can help you notice if you are missing certain kinds of requirements. In this example, it helped identify a lot of requirements that might have been missed or hidden inside developer assumptions.

Common Requirements

The following list summarizes some specific requirements that arise in many applications.

- **Screens**—What screens are needed?
- **Menus**—What menus will the screens have?

- **Navigation**—How will the users navigate through different parts of the system? Will they click buttons, use menus, or click forward and backward arrows? Or some combination of those methods?
- **Work flow**—How does data (work orders, purchase requests, invoices, and other data) move through the system?
- **Login**—How is login information stored and validated? What are the password formats (such as, must require at least one letter and number) and rules (as in, passwords must be changed monthly)?
- **User types**—Are there different kinds of users such as order entry clerk, shipping clerk, supervisor, and admin? Do they need different privileges?
- **Audit tracking and history**—Does the system need to keep track of who made changes to the data? (For example, so you can see who changed a customer to premier status.)
- **Archiving**—Does the system need to archive older data to free up space in the database? Does it need to copy data into a data warehouse for analysis?
- **Configuration**—Should the application provide configuration screens that let the system administrators change the way the program works? For example, those screens might let system administrators edit product data, set shipping and handling prices, and set algorithm parameters. (If you don't build these sorts of screens, you'll have to make those changes for the customers later.)

GATHERING REQUIREMENTS

At this point you know what makes a good requirement (clear, unambiguous, consistent, prioritized, and verifiable). You also know how to categorize requirements using audience-oriented, FURPS, or FURPS+ methods. But how do you actually pry the requirements out of the customers?

The following sections describe several techniques you can use to gather and refine requirements.

Listen to Customers (and Users)

Sometimes, customers come equipped with fully developed requirements spelling out exactly what the application should do, how it should work, and what it should look like. More often they just have a problem that they want solved and a vague notion that a computer might somehow help.

Start by listening to the customers. Learn as much as you can about the problem they are trying to address and any ideas they may have about how the application might solve that problem. Initially, focus as much as possible on the problem, not on the customers' suggested solutions, so you can keep the requirements flexible.

If the customers insist on a particular feature that you think is unimportant, or if they request something that just seems strange, ask them why they want it. Sometimes, the requirement may be a random thought that isn't actually important, but sometimes the customers have a good reason that you just don't understand. Often the reason is so obvious to them that it doesn't occur to them to explain it until you ask. The customers probably know a lot more about their business than you do, and they may make assumptions about facts that are common knowledge to them but mysterious to you.

AN OFFER YOU CAN'T REFUSE

Suppose The Don's Waste Removal Service asks you to write an application that lets users plot out routes for garbage trucks. You're working through the list of requirements with the owner, Don, and he says, "A route that contains lots of left turns should be given no respect."

To most people, that may seem like a strange requirement. What has Don got against left turns?

Don's been working with garbage trucks for a long time so, like many people who do a lot of vehicle routing, he knows that trucks turning left spend more time waiting for cross traffic, so they burn more fuel. They are also more likely to be involved in accidents. (It always amazes me that people can fail to notice a 20-ton garbage truck stopped in front of them, but it happened in my neighborhood not long ago.) Penalizing routes that contain left turns (and U-turns) will save the company money.

Take lots of notes while you're listening to the customers. They sometimes mention these important but puzzling tidbits in passing. If a customer requirement seems odd, dig a bit deeper to find out what, if anything, is behind the request.

Use the Five Ws (and One H)

Sometimes customers have trouble articulating their needs. You can help by using the five Ws (who, what, when, where, and why) and one H (how).

Who

Ask who will be using the software and get to know as much as you can about those people. Find out if the users and the customers are the same and learn as much about the users as you can.

For example, if you're writing medical billing software, the users might be data entry operators who type in patient data all day. In contrast, your customers may be corporate executives. They may have worked their way up through the ranks (in which case they probably know everything about medical data entry down to the last billing code) or they may have followed a more business-school-oriented career path (in which case they may not know a W59.22 from a V95.43). (It's worth the time to look these up in your favorite browser.)

What

Figure out what the customers need the application to do. Focus on the goals as much as possible rather than the customers' ideas about how the solution should work. Sometimes, the customers have good ideas about what the application should look like, but you should try to keep your options open. Often the project members have a better idea than the customers of the kinds of things an application can do, so they may come up with better solutions if they focus on the goals.

(Of course, the customer is always right, at least until your paycheck is signed, so if the customer absolutely insists that the application must include a graphical slide rule instead of a calculator, chalk it up as an interesting exercise in graphics programming and make it happen.)

When

Find out when the application is needed. If the application will be rolled out in phases, find out which features are needed when.

When you have a good idea about what the project requires, use Gantt charts and the other techniques described in Chapter 3, "Project Management," to figure out how much time is actually needed. Then compare the customers' desired timeline to the required work schedule. If the two don't match, you need to talk to the customers about deferring some features to a later release.

Don't let the customers assume they can get everything on their time schedule just by "motivating you harder." In *Star Trek*, Scotty can squeeze eight weeks' worth of work into just two, but that rarely works in real-world software engineering. You're far more likely to watch helplessly as your best programmers jump ship before your project hits the rocky shoals of impossible deadlines.

Where

Find out where the application will be used. Will it be used on desktop computers in an air-conditioned office? Or will it be used on phones in a noisy subway?

Why

Ask why the customers need the application. Note that you don't need to be unnecessarily stupid. If the customers say, "We want to automate our parts ordering system so that we can build custom scooters more quickly," you don't need to respond with, "Why?" The customers just told you why.

Instead, use the "why" question to help clarify the customers' needs and see if it is real. Sometimes, customers don't have a well-thought-out reason for building a new system. They just think it will help but don't actually know why. (Or customers may have just received a new copy of *Management Buzzwords Monthly* and they're convinced they can crowdsource custom scooter design.)

Find out if there is a real reason to believe a new application will help. Is the problem really that ordering parts is inefficient? Or is the problem that each order requires a different set of parts that have a long shipping time? If streamlining the ordering process will cut the ordering time from 2 days to 1.5 days, while still leaving 4–6 weeks of shipping delay, then a new software application may not be the best place to spend your resources. (It might be better to maintain an inventory of slow-to-order parts such as wheel spinners and spoilers.)

How

The "What" section earlier in this chapter said you should focus on the goals rather than the customers' ideas about the solution. That's true, but you shouldn't completely ignore the customers' ideas. Sometimes, customers have good ideas, particularly if they relate to existing practices. If the users are used to doing something a certain way, you may reduce training time by making the application mimic that approach. Be sure to look outside the box for other solutions, but don't automatically think that software developers always make better decisions than the customers.

Study Users

Interviewing customers (and users) can get you a lot of information, but often customers (and users) won't tell you everything they do or need to do. They often take for granted details that they consider trivial but that may be important to the development team.

For example, suppose the users grind through long, tedious reports every day. The reports are so long, they often end the day in the middle of a report and need to continue working on it the next day. This may seem so obvious to the users that you don't discuss the issue.

A typical reporting application might require the users to log in every day, search for a particular report, and double-click it to open it. That could take a while (particularly if the user forgets which report it is). Fortunately, you know that users often start the day by reopening the last report of the previous day, so you can streamline the process. Instead of making users remember what report they last had open, the program can remember. You can then provide a button or menu item to immediately jump to that report.

By studying users as they work, you can learn more about what they need to do and how they currently do it. Then with your software-engineering perspective, you can look for solutions that might not occur to the users.

PRINTING PUZZLE

Watching users in their natural habitat often pays off. Many years ago, I was visiting a telephone company billing center in preparation for a project that automatically identified customers who hadn't paid their bills and so it could disconnect their service. We spent a week there studying the existing software systems and the users. It was interesting, but the reason I'm mentioning it now is a small comment made by one of the managers. In passing, she said something like, "I sure wish you could do something about the Overdue Accounts Report. Ha, ha."

That's the sort of comment that should make you dig deeper. What was this report and why was it a problem? It turned out that the existing software system printed out a list of every customer with an outstanding balance for every billing cycle. This was a ***big*** billing center serving approximately 15 million customers, so every two days (there were 15 billing cycles per month) the printer spit out a 3-foot tall pile of paper listing every customer in the cycle with an outstanding balance.

Balances ranged from a few cents to tens of thousands of dollars, and the big-balance customers were costing the company tons of money. Unfortunately, the printout listed the customers in some weird arrangement (sorted by customer ID or zodiac sign or something), so the billing people couldn't find the customers with the big balances.

What the customers didn't know (but we did) is that it's relatively easy to build a printer emulation program. It took approximately one week (mostly spent getting management approval) to write a program that pretended to be a printer, sucked up all the overdue account information, and sorted it by balance. It turned out that of

the thousands of pages of data produced every two days, the customers only needed the first two.

The moral of the story is, you need to pay attention to the customers' comments. They don't know what you can do with the computer, and you don't know their needs.

As you study the users, pay attention to how they do things. Look at the forms they fill out (paper or online). Figure out where they spend most of their time. Look for the tasks that go smoothly and those that don't. You can use that information to identify areas in which your project can help.

REFINING REQUIREMENTS

After you've talked to the customers and users, and watched the users at work, you should have a good understanding about the users' current operations and needs. (If you don't, ask more questions and watch the users some more until you do.)

Next, you need to use what you've learned to develop ideas for solving the user's problems. You need to distill the goals (what the customers need to do) into approaches (how the application will do it).

At a high level, the requirement, "Process customer records" is fine. It's also nice and flexible, so it allows you to explore many options for achieving that goal.

At some point, however, you need to turn the goals into something that you can actually build. You need to figure out how the users will select records to edit, what screens they will use, and how they will navigate between the screens. Those decisions will lead to requirements describing the forms, navigation techniques, and other features that the application must provide to let the users do their jobs.

NOTE *Moving from goals to requirements often forces you to make some design decisions. For example, you may need to specify form layouts (at least roughly) and the way work flows through the system.*

You might think of those as design tasks, but they're really part of requirements gathering. The following two chapters, which talk about design, deal with program design (how you structure the code) not user interface design and the other sorts of design described here.

The following two sections describe three approaches for converting goals into requirements.

Copy Existing Systems

If you're building a system to replace an existing system or a manual process, you can often use many of the behaviors of the existing system as requirements for the new one. If the old system sends customers e-mails on their birthdays, you can require that the new system does that,

too. If the users currently fill out a long paper form, you can require that the new system has a computerized form that looks similar—possibly with some tabs, scrolled windows, and other format changes to make the form look a bit better on a computer.

This approach has a few advantages. First, it's reasonably straightforward. It doesn't take an enormous amount of software engineering experience to dig through an existing application and write down what it does. (If you're lucky, you might even get the customers to do at least some of it so that you can focus on software design issues.)

This approach also makes it more likely that the requirements can actually be satisfied, at least to the extent the current system works. If an existing system does something, then you at least know it's possible.

Finally, this approach provides an unambiguous example of what you need to do. In the specification, you don't need to write out in excruciating detail exactly how the "Lazy Backup" screen works. Instead you can just say, "The Lazy Backup screen will work as it does in the existing system with the following changes: ...."

Even though this approach is straightforward, it has some disadvantages. First, you probably wouldn't be building a new version of an existing system unless you planned to make some changes. Those changes aren't part of the original system, so there's no guarantee that they're even possible. They may also be incompatible with the original system. (Not all pieces of software play nicely together.)

A second problem with this approach is that users are often reluctant to give up even the tiniest features in an existing program. In the projects I've worked on, I've found that no matter how obscure and worthless a feature is, there's at least one user willing to fight to the death to preserve it. If the software has been in use for a long time, it may contain all sorts of odd quirks and peccadillos. You might like to streamline the new system by removing the feature that changes the program's background color to match the weather each day, but that's not always possible.

FOREVER FEATURES

I was once asked to help port part of an application to a new platform. The key piece of the application that the customer wanted to keep was fairly small, and the project manager estimated it would take a few hundred hours of work to get the job done.

When I dug through the original application, however, I found that it included more than 100 forms, each of which was moderately complicated. The system also included interfaces to a number of external databases and automated systems.

At this point, we went back to the customer and asked if they were willing to give up most of those 100+ forms and just keep the key tools we were trying to port.

By now you've probably guessed the punchline. The customer wouldn't give up any of the existing application's features. The project's estimated time jumped from a few hundred hours to several thousand hours, and the whole thing was scrapped.

There is some good news in this tale, however. We discovered the problem quickly during initial requirements gathering, so we hadn't wasted too much time before the project was canceled. It would have been much worse if we had started work only to have the requirements gradually expand to include everything in the original application. Then we would have wasted hundreds of hours of work before the project was canceled.

Using an existing system to generate requirements can be a big time-saver, as long as the development team and the customers all agree on which parts of the existing system will be included in the new one.

Clairvoyance

A lot more often than you might think, one or more people simply look at the project's goals, visualize a finished result, and start cranking out requirements. For example, the project lead might use gut feelings, common sense, tea leaves, tarot cards, and other arcane techniques to cobble together something that he thinks will work. If the project is large, pieces might be doled out to team leads so that they can work on their own pieces of the system, but the basic approach is the same: Someone sits down and starts churning out form designs, work flow models, login procedures, and descriptions of reports.

I'm actually being a bit unfair characterizing this approach as clairvoyance because it's actually quite effective in practice. Assuming the people writing the requirements understand the customers' needs and have previous experience, they often produce a good result. Ideally team leads are chosen for their experience and technical expertise (not because they're the boss's cousin), so they know what the computer can do and they can design a system that works.

This technique is particularly effective if the project lead has previously built a similar system. In that case, the lead already knows more or less what the application needs to do, which things will be easy and which will be hard, how much time everything requires, and which kinds of donuts motivate the programmers the best.

Having an experienced project lead greatly increases the chances that the requirements will include everything you need to make the project succeed. It also greatly increases the chances that the team will anticipate problems and handle them easily as development continues. In fact, this is such an important point, it's a best practice.

BEST PRACTICE: EXPERIENCED PROJECT LEADS

A project's chances for success are greatly improved if the project lead has previous experience with the same kind of project.

The same holds true for the other project members. Programmers with previous experience with the same kind of project will encounter fewer problems and meet their scheduled milestones more often.

Documenters who have written user manuals for similar applications will find writing manuals for the new project easier. Project managers with similar experience will know what tasks are likely to be difficult. Even customers with previous software engineering experience will be better at creating good requirements.

If you have access to design specialists such as user interface designers or human factors experts, get them to help. Any programmer can build forms, menus, and colorful labels, but some don't do a good job. A good user interface makes users productive. A bad one is frustrating and ineffective. (It's like trying to empty a bathtub with a teaspoon. You'll eventually succeed, but you'll spend the whole time thinking, "This is stupid. There has to be a better way!")

Brainstorm

Copying an existing application and clairvoyance are good techniques for generating requirements, but they share a common disadvantage: They are unlikely to lead you to new innovative solutions that might be better than the old ones. To find truly revolutionary solutions, you need to be more creative. One way to look for creative solutions is the group creativity exercise known as *brainstorming*.

You're probably somewhat familiar with brainstorming, at least in an informal setting, but there are several approaches that you can use under different circumstances.

The basic approach that most people think of as brainstorming is called the *Osborn method* because it was developed by Alex Faickney Osborn, an advertising executive who tried to develop new, creative problem-solving methods starting in 1939. Basically, he was tired of his employees failing to come up with new and innovative advertising campaigns. (As is the case with the Gantt charts described in Chapter 3, the fact that we're still using Osborn's techniques after all these years shows how useful they are.) Osborn's key observation is summed up nicely in his own words.

> *It is easier to tone down a wild idea than to think up a new one.*
>
> —Alex Faickney Osborn

Basically, the gist of the method is to gather as many ideas as possible, not worrying about their quality or practicality. After you assemble a large list of possible ideas, you examine them more closely to see which deserve further work.

To allow as many approaches as possible, you should try to get a diverse group of participants. In software engineering, that means the group should include customers, users, user interface designers, system architects, team leads, programmers, trainers, and anyone else who has an interest in the project. Get as many different viewpoints as you can. (Although in practice brainstorming becomes less effective if the group becomes larger than 10 or 12 people.)

To keep the ideas flowing, don't judge or critique any of the ideas. If you criticize someone's ideas, that person may shut down and stop contributing. Even a truly crazy idea can spark other ideas that may lead somewhere promising. Just write down every idea no matter how impractical it may seem. Even if an idea is impossible to implement using today's technology, it may be simple by next Wednesday.

(It wasn't that long ago that portable phones had the size, weight, and functionality of a brick. Now they're small enough to lose in the sofa cushions and have more computing power than NASA had when Neil Armstrong flubbed his "one small step" line on the moon.)

Osborn's method uses the following four rules:

1. **Focus on quantity.** Do everything you can to keep the ideas flowing. The more ideas you collect, the greater your chances of finding a really creative and revolutionary solution.
2. **Withhold criticism.** Criticism can make people stop contributing. Early criticism can also eliminate seemingly bad ideas that lead to better ideas.
3. **Encourage unusual ideas.** You can always "tone down a wild idea" but you may need to think way outside of the box to find really creative solutions.
4. **Combine and improve ideas.** Form new ideas by combining other ideas or using one idea to modify another.

Only after the flow of ideas is slowing to a trickle should you start evaluating the ideas to see what you've got. At that point, you can pick out the most promising ideas to develop further (possibly with more brainstorming).

Many people are familiar with Osborn's method (although they may not know its name), but there are also several other brainstorming techniques, some of which can be even more effective. The following list describes some of those techniques.

- **Popcorn**—(I think of this as the Mob technique.) People just speak out as ideas occur to them. This works fairly well with small groups of people who are comfortable with each other.
- **Subgroups**—Break the group into smaller subgroups (possibly in the same room) and have each group brainstorm. When the subgroups are finished, have the larger group discuss their best ideas. This works well if the main group is very large, if some people feel uncomfortable speaking in the larger group (the new developer in shorts and sandals may be afraid to speak out in front of the corporate vice president in a thousand dollar suit), or if one or two people are monopolizing the discussion.
- **Sticky notes**—Also called the nominal group technique (NGT). Participants write down their ideas on sticky notes, index cards, papyrus, or whatever. The ideas are collected, read to the group, and the group votes on each idea. The best ideas are developed further, possibly with other rounds of brainstorming.
- **Idea passing**—Participants sit in a circle. (I suppose you could use some other arrangement such as an ellipse, rectangle, or nonagon. As long as you have an ordering for the participants.) Each person writes down an idea and passes it to the next person. The participants add thoughts to the ideas they receive and pass them on to the next person. The ideas continue moving around the circle until everyone gets their original idea back. At this point, each idea should have been examined in great detail by the group. (Instead of a circle, nonagon, or whatever, you can also swap ideas randomly.)
- **Circulation list**—This is similar to idea passing except the ideas are passed via e-mail, envelope, or some other method outside of a single meeting. This can take a lot longer than idea passing but may be more convenient for busy participants.
- **Rule breaking**—List the rules that govern the way you achieve a task or goal. Then everyone tries to think of ways to break or circumvent those rules while still achieving the goal.

- **Individual**—Participants perform their own solitary brainstorming sessions. They can write (or speak) their trains of thought, use word association, draw mind maps (diagrams relating thoughts and ideas—search online for details), and any other technique they find useful. Some studies have shown that individual brainstorming may be more effective than group brainstorming.

The following list describes some tips that can make brainstorming more productive.

- Work in a comfortable room where everyone can feel at ease.
- Provide food and drinks. (I'll let you decide what kinds of drinks.)
- Start by recapping the users' current processes and the problems you are trying to solve.
- Use a clock to keep sessions short and lively. If you're using an iterative approach such as idea passing, keep the rounds short.
- Allow the group's attention to wander a bit, but keep the discussion more or less on topic. If you're designing a remote mining rig control system, then you probably don't need to be discussing Ouija boards or Monty Python quotes.
- However, a few jokes can keep people relaxed and help ideas flow, so a few Monty Python quotes may be okay.
- If you get stuck, restate the problem.
- Allow silent periods so that people have time to think about the problem and their ideas.
- Reverse the problem. For example, instead of trying to think of ways to build better blogging software, think of ways to build worse blogging software. (Obviously, don't actually do them.)
- Write ideas in slightly ambiguous ways and let people give their interpretations.
- At the end, summarize the best ideas and give everyone copies so that they can think about them later. Sometimes, a great idea pops into someone's head after the official brainstorming sessions are over.

Brainstorming is useful any time you want to find creative solutions to complex problems, not just during requirements gathering. You can use it to pick problems in your company that you might solve with a new software project. You can use it to design user interfaces, explore possible system architectures, create high-level designs, and plan interesting exercises for training classes. (You can even use brainstorming techniques outside of software engineering to decide where to go on your next vacation, reduce pollution in your city, or pick a school science fair project.)

Keep brainstorming in mind throughout the project as a technique you can use to attack difficult problems.

RECORDING REQUIREMENTS

After you decide what should be in the requirements, you need to write them down so that everyone can read them (and argue about whether they're correct). There are several ways you can record requirements so team members can refer to them throughout the project's lifetime.

Obviously, you can just write the requirements down as a sequence of commandments as in, "Thou shalt make the user change passwords on every full moon." There's a lot to be said for writing down requirements in simple English (or whatever your team's native language is). For starters, the team members already know that language and have been using it for many years.

You can still mess things up by writing requirements ambiguously or in hard-to-understand formats (such as limericks or haiku), but if you're reasonably careful, requirements written in ordinary language can be very effective.

The following sections describe some other methods for recording requirements.

UML

The Unified Modeling Language (UML) lets you specify how parts of the system should work. Despite its name, UML isn't a single unified language. Instead it uses several kinds of diagrams to represent different pieces of the system. Some of those represent program items such as classes. Others represent behaviors, such as the way objects interact with each other and the way data flows through the system.

I won't bash UML (it's too popular and I'm not famous enough to get away with it), but it does have some drawbacks. Most notably it's complicated. UML includes two main categories of diagrams that are divided into more than a dozen specific types, each with its own complex set of rules.

Specifying complex requirements with UML is only useful if everyone understands the UML. Unfortunately, many customers and users don't want to learn it. It's not that they couldn't. They just usually have better things to do with their time, like helping you understand their needs. (I did actually work on one project where the customers taught themselves how to use some types of UML diagrams so that they could specify parts of the system. It worked reasonably well, but it took a long time.)

I'll talk more about UML in the next chapter. For now during requirement gathering, you probably shouldn't rely heavily on UML unless your customers are already reasonably familiar with it. (For example, if you're writing a library for use by other programmers who already use UML.)

User Stories

> *Storytelling strikes me as a more powerful tool than quantification or measurement for what we do.*
>
> —Alan Cooper

A ***user story*** is exactly what you might think: a short story explaining how the system will let the user do something. For example, the following text is a story about a user searching a checkers database to find opponents:

> *The user enters his Harkness rating (optional), whether moves should be timed or untimed, and the variant (such as traditional, three-dimensional, upside-down, or Gliński). When the user clicks Search, the application displays a list of possible opponents that have compatible selections.*

Many developers write stories on index cards to encourage brevity. The scope of each story should also be limited so that no story should take too long to implement (no more than a week or two).

Notice that the story doesn't contain a lot of detail about things like whether the game variants are given in a list or set of radio buttons. The story lets you defer those decisions until later during design.

User stories should come with acceptance testing procedures that you can use at the end of development to decide whether the application satisfied the story.

User stories may seem low-tech, but they have some big advantages, not least of which is that people are already familiar with them. They are easy to write, easy to understand, and can cover just about any situation you can imagine. They can be simple or complex depending on the situation. Unlike UML, your customers, developers, managers, and other team members already know how to understand stories without any new training.

User stories give you a lot of expressiveness and flexibility without a lot of extra work. (In management speak, user stories allow you to leverage existing competencies to empower stakeholders.)

User stories do have some drawbacks. For example, you can easily write stories that are confusing, ambiguous, inconsistent with other stories, and unverifiable. Of course, that's true of any method of recording requirements.

Use Cases

A *use case* is a description of a series of interactions between actors. The actors can be users or parts of the application.

Often a use case has a larger scope than a user story. For example, a use case might explain how the application will allow a user to examine cardiac ultrasound data for a patient. That user might need to use many different screens to examine different kinds of recordings and measurements. Each of those subtasks could be described by a user story, but the larger job of examining all the data would be too big to describe on a single index card and would take longer to implement than a week or two.

Use cases also follow a template more often than user stories. A simple template might require a use case to have the following fields:

- **Title**—The name of the goal as in, "User Examines Cardiac Data." Usually, the title includes an action (examines) and the main actor (user).
- **Main success scenario**—A numbered sequence of steps describing the most normal variation of the scenario.
- **Extensions**—Sequences of steps describing other variations of the scenario. This may include cases such as when the user enters invalid data or the application can't handle a request. (For example, if the user searches for a nonexistent patient.)

Other templates include a lot more fields such as lists of stakeholders interested in the scenario, preconditions that must be met before the scenario begins, and success and failure variations.

Prototypes

A *prototype* is a mockup of some or all of the application. The idea is to give the customers a more intuitive hands-on feel for what the finished application will look like and how it will behave than you can get from text descriptions such as user stories and use cases.

A simple user interface prototype might display forms that contain labels, text boxes, and buttons showing what the finished application will look like. In a *nonfunctional prototype*, the buttons, menus, and other controls on the forms wouldn't actually do anything. They would just sit there and look pretty.

A *functional prototype* (or *working prototype*) looks and acts as much like the finished application will but it's allowed to cheat. It may do something that looks like it works, but it may be incomplete and it probably won't use the same methods that the final application will use. It might use less efficient algorithms, load data from a text file instead of a database, or display random messages instead of getting them from another system. It might even use hard-coded fake data.

For example, the prototype might let you enter search criteria on a form. When you clicked the Search button, the prototype would ignore your search criteria and display a prefilled form showing fake results. This gives the customers a good idea about how the final application will work but it doesn't require you to write all the code.

There are a couple of things you can do with a prototype after it's built. First, you can use it to define and refine the requirements. You can show it to the customers and, based on their feedback, you can modify it to better fit their needs.

After you've fine-tuned the prototype so that it represents the customers' requirements as closely as possible, you can leave it alone. You can continue to refer to it if there's a question about what the application should look like or how it should work, but you start over from scratch when building the application. This kind of prototype is called a *throwaway prototype*.

Alternatively, you can start replacing the prototype code and fake data with production-quality code and real data. Over time, you can evolve the prototype into increasingly functional versions until eventually it becomes the finished application. This kind of prototype is sometimes called an *evolutionary prototype*. This approach is used by some of the iterative approaches described in Chapter 13.

SURVIVAL OF THE LAZIEST

You need to be careful if you use an evolutionary prototype. While throwing together an initial version to show the customers what the final application will do, developers can (and should) take shortcuts to get things done as quickly as possible. That can result in code that's sloppy, riddled with bugs, and hard to maintain.

As long as the prototype works, that's fine. The prototype is only supposed to give you an idea about how the program will work, so it doesn't need to be as maintainable as the finished application in the long run.

That's fine if you're using the prototype only to define requirements, but if you try to evolve the prototype into a production application, you need to be sure to go back and remove all the shortcuts and rewrite the code properly. If you don't remove all the prototype code, you'll certainly pay the price later in increased bug fixes and maintenance.

Requirements Specification

How formally you need to write up the requirements depends on your project. If you're building a simple tool to rename the files on your own computer in bulk, a simple description may be enough. If you're writing software to fill out legal forms for a law firm, you probably need to be much more formal. (And you might want to hire a different law firm to review your contract.)

If you search the Internet, you can find several templates for requirement specifications. These typically list major categories of requirements such as user documentation, user interface design, and interfaces with other systems.

For example, Villanova University has an example template in Word format at `tinyurl.com/obqhatt`. The North Carolina Enterprise Project Management Office has another one at `tinyurl.com/n7ttqfh`. (These are really long URLs so I used tinyurl to shorten them.)

VALIDATION AND VERIFICATION

After you record the requirements (with whatever methods you prefer), you still need to validate them and later verify them. The two terms validation and verification are sometimes used interchangeably. Here are the definitions I use. (I think these are the most common interpretations.)

Requirement validation is the process of making sure that the requirements say the right things. Someone, often the customers or users, need to work through all the requirements and make sure that they: (1) Describe things the application should do. (2) Describe *everything* the application should do.

Requirement verification is the process of checking that the finished application actually satisfies the requirements.

VALIDATION VERSUS VERIFICATION

Another way to think of this is

Validation—Are we doing the right things?

Verification—Are we doing the things right?

Those two statements are glib, but it's hard to remember which is which. Perhaps a better way to remember the difference is that "validation" comes before "verification" alphabetically and validation comes before verification in a software project.

CHANGING REQUIREMENTS

In many projects, requirements evolve over time. As work proceeds, you may discover that something you thought would be easy is hard. Or you may stumble across a technique that lets you add a high-value feature with little extra work.

Often changes are driven by the customers. After they start to see working pieces of the application, they may think of other items that they hadn't thought of before.

Depending on the kind of project, you may accommodate some changes, as long as they don't get out of hand. You can help control the number of changes by creating a *change control board*. Customers (and others) can submit change requests to this board (which might actually be a single person) for approval. The board decides whether a change should be implemented or deferred to a later release.

The development methods described in Chapters 13 and 14 are particularly good at dealing with changing requirements because they tend to build an application in small steps with frequent opportunities for refinement. If you add new features in a mini-project every two weeks, it's easy to add new requirements into the next phase. There's still a danger of never finishing the project, however, if the change requests keep trickling in.

SUMMARY

Requirements gathering may not be *the most* important stage of a project, but it certainly is *an* important stage. It sets the direction for future development. If you get the requirements wrong, you may develop something but there's no guarantee that it will be useful.

Good requirements must satisfy some basic requirements of their own. For example, they must be clear and consistent. Having hundreds of requirements won't do you any good if no one can understand what they mean or if they contradict each other.

Some developers group requirements into categories. For example, you can use audience-oriented categories, FURPS, or FURPS+ to organize requirements. Categorizing requirements alone doesn't help you define the project, but you can use the categories as a checklist to make sure you haven't forgotten anything obvious. (They also make it easier to understand other software engineers at parties when they say, "This party has good functional requirements but the nonfunctionals could use some work!")

There are several ways you can gather requirements. Obviously, you should talk with the customers and, if possible, the users. You can use the five Ws and one H to help guide the conversation. Studying the users as they currently perform their jobs is often instructive. It can help clarify the project's goals, and occasionally you may discover simple things you can add to the project that will make the users' jobs a whole lot easier.

After you understand the customers' needs, you must refine those needs into requirements. Three techniques that can help include: copying an existing system, using previous experience to just write them down, and brainstorming. Brainstorming is often more work but can sometimes lead to creative solutions that you might not have discovered otherwise.

Finally, after you know what the requirements are, you need to record them so everyone can refer to them as the project continues. Some ways you can record requirements include formal written specifications, UML diagrams, user stories, use cases, and prototypes.

Before you move on to the next phase of development, you should validate the requirements to ensure that they actually meet the customers' needs. (Later, near the end of the project, you'll also need to verify that the project has met the requirements.)

If you think this seems like a lot of work before the project "actually" begins, you're right. However, it's critical to the project's eventual success. Without sound requirements, how will you know what

to build? Unless the requirements are clear and verifiable, how will you know if you've achieved your goals?

After you gather, record, and validate the requirements, you're ready to move on to the next stage of development: high-level design. You may have already incorporated some design decisions into the requirements. For example, you might have made some user interface decisions or picked a system architecture.

The next chapter explains some of the high-level design decisions you might need to make, whether in the requirements phase or during a separate high-level design step. It also provides some guidance on how to make those decisions.

EXERCISES

1. List five characteristics of good requirements.
2. What does MOSCOW stand for?
3. Suppose you want to build a program called TimeShifter to upload and download files at scheduled times while you're on vacation. The following list shows some of the application's requirements.
   a. Allow users to monitor uploads/downloads while away from the office.
   b. Let the user specify website log-in parameters such as an Internet address, a port, a username, and a password.
   c. Let the user specify upload/download parameters such as number of retries if there's a problem.
   d. Let the user select an Internet location, a local file, and a time to perform the upload/download.
   e. Let the user schedule uploads/downloads at any time.
   f. Allow uploads/downloads to run at any time.
   g. Make uploads/downloads transfer at least 8 Mbps.
   h. Run uploads/downloads sequentially. Two cannot run at the same time.
   i. If an upload/download is scheduled for a time when another is in progress, it waits until the other one finishes.
   j. Perform scheduled uploads/downloads.
   k. Keep a log of all attempted uploads/downloads and whether they succeeded.
   l. Let the user empty the log.
   m. Display reports of upload/download attempts.
   n. Let the user view the log reports on a remote device such as a phone.

o. Send an e-mail to an administrator if an upload/download fails more than its maximum retry number of times.

p. Send a text message to an administrator if an upload/download fails more than its maximum retry number of times.

For this exercise, list the audience-oriented categories for each requirement. Are there requirements in each category?

4. Repeat Exercise 3 using the FURPS requirement categories.

5. What are the five Ws and one H?

6. List three techniques for gathering requirements from customers and users.

7. Explain why brainstorming can be useful in defining requirements.

8. List the four rules of the Osborn method.

9. Figure 4-1 shows the design for a simple hangman game that will run on smartphones. When you click the New Game button, the program picks a random mystery word from a large list and starts a new game. Then if you click a letter, either the letter is filled in where it appears in the mystery word, or a new piece of Mr. Bones's skeleton appears. In either case, the letter you clicked is grayed out so that you don't pick it again. If you guess all the letters in the mystery word, the game displays a message that says, "Congratulations, you won!" If you build Mr. Bones's complete skeleton, a message says, "Sorry, you lost."

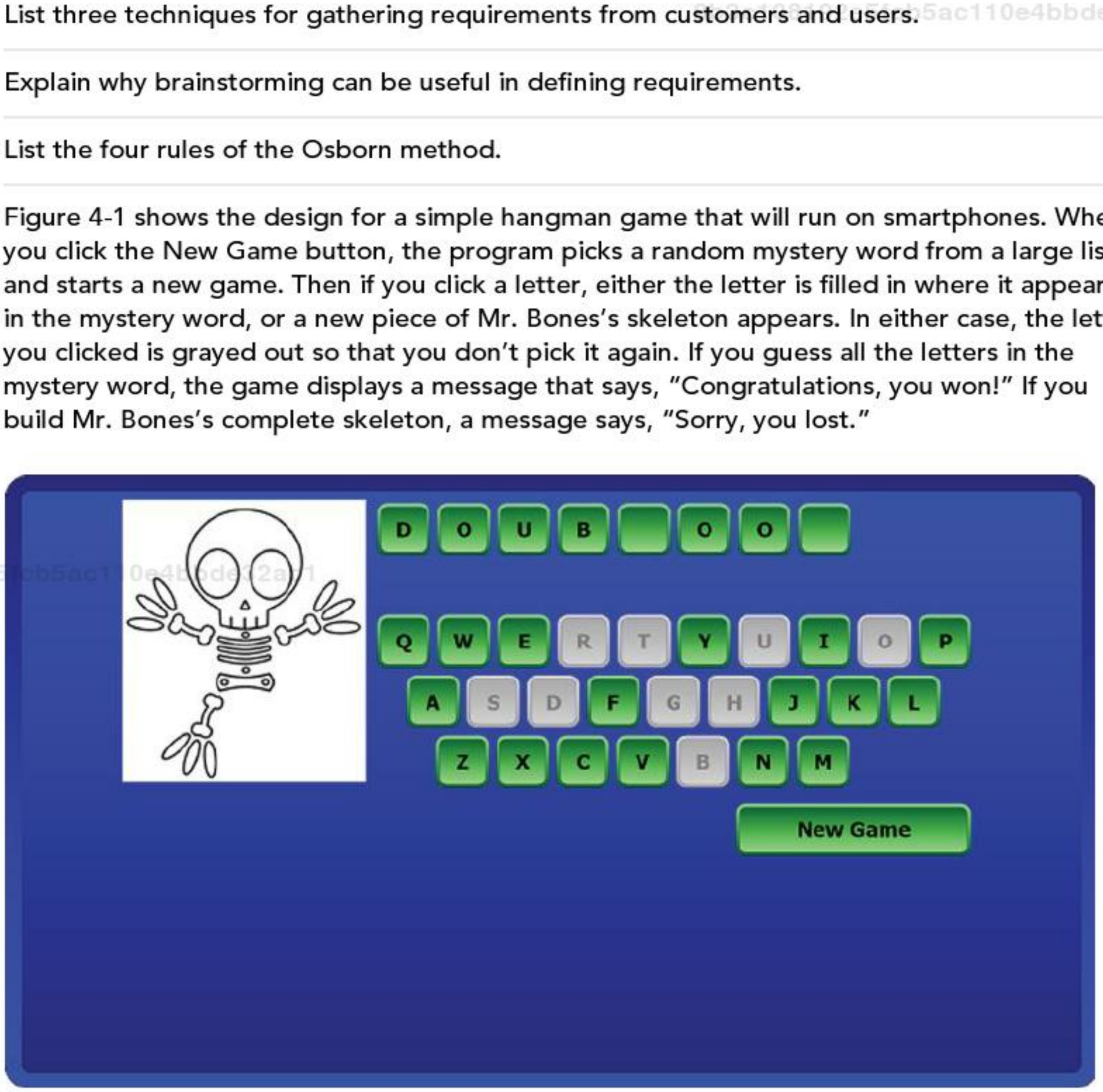

FIGURE 4-1: The Mr. Bones application is a hangman word game for Windows Phone.

Brainstorm this application and see if you can think of ways you might change it. Use the MOSCOW method to prioritize your changes.

10. (Instructors) Have the class brainstorm ideas to address a fairly difficult issue (such as reversing global warming, ending global hunger, or making politicians honor their campaign promises). If time permits, try a couple different brainstorming variations such as popcorn, subgroups, and individual. Discuss what went well and what didn't.

▸ WHAT YOU LEARNED IN THIS CHAPTER

- Requirements are important to a project because they set the project's goals and direction.
- Requirements must be clear, unambiguous, consistent, prioritized, and verifiable.
- The MOSCOW method provides one way to prioritize requirements.
- FURPS stands for Functionality, Usability, Reliability, Performance, and Supportability.
- FURPS+ also adds design constraints, implementation requirements, interface requirements, and physical requirements.
- You can gather requirements by talking to customers and users, watching users at work, and studying existing systems.
- You can convert goals into requirements by copying existing systems and methods, using previous experience to write them down, and brainstorming.
- You can record requirements in written specifications, UML diagrams, user stories, use cases, and prototypes.
- Requirements may change over time. That's okay as long as it happens in a controlled manner.
- Requirement validation is the process of checking that the requirements meet the customers' needs.
- Requirement verification is the process of checking that the finished project satisfies the requirements.