9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

8 Testing

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Testing is the process of comparing the invisible to the ambiguous, so as to avoid the unthinkable happening to the anonymous.

—James Bach

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Goals of testing
- Reasons why you might not want to remove a bug
- How to prioritize bugs
- Kinds of tests and testing techniques
- Good testing habits
- Methods for estimating number of bugs

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

It's a software engineering axiom that all nontrivial programs contain bugs. Actually, it's such an important point that it deserves to be put in its own note and explained with an example.

> **NOTE** *All Nontrivial Programs Contain Bugs*

For example, Windows 2000 was said by some to contain a whopping 63,000+ known bugs when it was shipped. Microsoft quickly retorted that this number didn't actually

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

count bugs. It included feature requests, notes asking developers to make something work better or more efficiently than it already did, clarification requests, and other nonbug issues. The *true* bugs, Microsoft explained, were mostly minor issues that wouldn't seriously hurt users.

No matter how you count them, Windows 2000 contained a *lot* of "undesirable features" ranging from changes that should probably have been made to indisputable bugs.

You might think that's the result of shoddy workmanship, but no project of that size could possibly have shipped without any bugs. The industry average number of bugs per thousand lines of code (kilo lines of code or *KLOC*) is typically estimated at about 15 to 50. When you consider that Windows 2000 contained more than 29 million lines of code, it's a miracle it works at all. Even assuming 10 bugs per KLOC, Windows 2000 should contain approximately 290,000 bugs. Suddenly 63,000 bugs is starting to look good, isn't it?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

NASA's Goddard Space Flight Center, which takes bugs *very* seriously because a mistake in its code could cost lives and hundreds of millions of dollars, is said to have reduced its number of bugs to less than 0.1 per KLOC. Even if Microsoft could afford to follow Goddard's practices (and getting the number of bugs per KLOC down to that level is *very* expensive), Windows 2000 would still contain 2,900 bugs.

I don't know about you, but if I encountered 2,900 bugs on a daily basis, I'd toss my computer out a window and start using a typewriter.

BIGGER AND BETTER?

Windows 8 is rumored to contain between 30 and 80 million lines of code, so clearly Microsoft isn't reducing its bug count by shrinking its operating system, but don't think this is just Microsoft's problem. The Firefox browser contains approximately 10 million lines of code, the Linux operating system contains more than 50 million, Mac OS X Tiger has approximately 85 million, and Facebook has approximately 60 million. It takes a lot of code to include something to annoy everyone!

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You can see an interesting chart showing the size of some big applications at `www.makeuseof.com/tag/million-lines-code-lot`.

Given that any nontrivial program contains bugs, what can you do about it? Are you doomed to suffer the slings and arrows of outraged customers? Or should you just throw in the towel and open a florist's shop instead of writing software?

Even though you can't wipe out every bug, you can catch the ones that will be most irritating to users. You can reduce the number of high-profile bugs to the point where users see them only rarely. If your program uses a good design, it should also recover from bugs gracefully so that the program doesn't crash.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

This chapter explains testing techniques you can use to flush out the majority of the most annoying bugs. It explains kinds of tests you should run and when to run them. It also explains how to estimate the number of bugs in the system so that you have some idea of whether you're getting closer to your goal of eliminating the high-profile bugs.

TESTING GOALS

Ideally you would sit down, write code that perfectly satisfies the requirements, and you'd be done. Unfortunately that rarely happens. More often than not, the first attempt at the software satisfies some but not all the requirements. It may also incorrectly handle situations that weren't specified in the requirements. For example, the code may not work in every possible situation.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

That's where testing comes in. Testing lets you study a piece of code to see whether it meets the requirements and whether it works correctly under all circumstances. (Usually, the second goal means a method works properly with any set of inputs.)

To get a complete picture of how a piece of code performs, you can carry out several different kinds of tests using a variety of techniques. Sections later in this chapter describe some of the most important of those. Before you get to them, however, it's worth knowing that it's not always worth removing every single bug from a program. Instead the goal is often to reduce the number bugs and their frequency of occurrence so that users can get their jobs done with a minimum of annoyance.

REASONS BUGS NEVER DIE

Simply put, a ***bug*** is a flaw in a program that causes it to produce an incorrect result or to behave unexpectedly. Bugs are generally evil (although occasionally they make games more fun), but it's not always worth your effort to try to remove every bug. Removing some bugs is just more trouble than it's worth. The following sections describe some reasons why software developers don't remove every bug from their applications.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Diminishing Returns

Finding the first few bugs in a newly written piece of software is relatively easy (and therefore cheap). After a few months of testing, finding bugs may become extremely difficult. At some point, finding the next bug would cost you more than you'll ever earn by selling the software.

Deadlines

In a just and fair world, software would be released when it was ready. In the real world, however, companies are often driven by deadlines imposed by management, competition, or a marketing department.

You might delay a release to fix high-profile bugs, but if the remaining bugs aren't too bad, you might be forced to release before you would like.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Consequences

Sometimes a bug fix might have undesirable consequences. For example, suppose you're building a drawing application and the tool that draws spirals isn't saving the spirals' colors correctly. You could fix it, but that would require changing the format of the saved picture files. That would force the users to convert their data files, and that would make them storm your office with torches and pitchforks.

In this example, it might be better to leave the spiral-saving code unfixed for now and include a fix in the next major release. Users expect some pain in major releases, so you might get away with it then. (But just in case, you should make sure the escape helicopter is fueled and ready to go.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

It's Too Soon

If you just released a version of a program, it may be too soon to give the users a new patch to fix a minor bug. Users won't like you if you release new bug fixes every 3 days. As a rule of thumb:

- If a bug is a security flaw, release a patch immediately, even if you just released a patch yesterday. (If you did release a patch yesterday, you better be sure the new patch fixes things correctly! Your reputation is at stake.) Include a note explaining how wonderful you are for protecting the users' valuable data.
- If a bug makes users swear at your program more than once a day, release a patch as soon as possible (as often as monthly). Include a profuse apology.
- If a bug is annoying enough to make users smirk at your program occasionally, fix it in a minor release (as often as twice a year). Include a huge fanfare about how great you are for looking after the users' needs.
- If a bug is just a nice-to-have new feature or a performance improvement, fix it in the next major release (at most once per year). Explain how responsive you are and that the users' needs are your number one concern.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Too many releases will annoy users, so you need to weigh the benefit of any bug patch against the inconvenience.

Usefulness

Sometimes users come to rely on a particular bug to do something sneaky that you didn't intend them to do. They won't thank you if you remove their favorite feature, even if it started out as a bug.

> *Any sufficiently advanced bug is indistinguishable from a feature.*
>
> —Bruce Brown

If the users have adopted a bug and are using it in their favor, formalize it and add it to the application's requirements. You may extend its behavior to give the users an even better feature and take credit for it.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Obsolescence

Over time, some features may become less useful. Eventually they may go away entirely. In that case, it may be better to just let the feature die rather than spending a lot of time fixing it.

For example, if an operating system has a bug in a floppy drive controller that limits its performance, it may be just as well to ignore it. Floppy drives are rare these days, so the bug probably won't inconvenience too many users. (In fact, some computers are shipping without CD or DVD drives these days. As long as your network and USB devices work, you may cut back on maintenance of your CD drivers. Of course, you still need to use a USB CD drive, at least for now.)

It's Not a Bug

Sometimes users think a feature is a bug when actually they just don't understand what the program is supposed to do. (It seems like Facebook has perfected this problem. It moves its security settings around and users complain that their cat pictures are visible to everyone in the world.)

This is really a problem of user education. Sometimes the documentation isn't correct and sometimes it's missing entirely. Sometimes the user isn't willing to read all the way through both paragraphs of documentation and see that the feature is clearly described.

If the documentation is incomplete or unclear, this is a "documentation bug" that you can fix the next time you release a new version of the documentation.

DOCUMENTATION DELETED

Back in the old days, when you bought a piece of software, you also got a nice, fat book explaining how to use it. Some particularly long user manuals came as a set of ring binders with pages that you could replace when the vendor sent you manual updates.

These days most application documentation comes electronically in text files, PDF files, or online help applications. That allows vendors to update the documentation any time it is necessary. (See the earlier section "It's Too Soon." The same ideas apply to documentation as well as software. Your customers won't thank you for sending them daily documentation updates.)

You can greatly decrease this problem by using a good user interface design. If the application groups features logically so users can find them easily, the users won't complain that a feature is missing when it isn't. If features are named clearly so it's obvious what they do, users won't complain that a feature doesn't do what they think it's supposed to do.

It Never Ends

If you try to fix every bug, you'll never release anything.

This is similar a problem you may have when buying a new computer. If you just wait another couple months, something faster will come out for the same price. If you do buy a nice, shiny, new machine, something better instantly goes on sale. At some point, you just need to pry open your

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

wallet, buy something, and get on with your life. (It also helps to avoid looking at advertisements afterwards, so you don't see the faster machines going on sale.)

Similarly at some point you need to stop testing, cross your fingers, and publish your application. It's almost guaranteed to be imperfect, but hopefully it's better than nothing.

It's Better Than Nothing

As the previous section mentions, your application may not be perfect, but hopefully it's better than nothing. In some cases, it may be so much better than nothing that it's worth releasing the application even though it's seriously flawed.

This is particularly true if your application is for in-house use. If you're writing a tool for your fellow employees to use, they may be willing to put up with some rough edges to get their jobs done more easily.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

THE TOOLSMITH

If a software project is large enough, it may be worth having a dedicated toolsmith. A *toolsmith* is someone whose job is to build tools for use by others on the project. A tool might count the lines of code in the project's modules, rearrange the controls on a form in top-down/left-to-right order, search customer data for patterns, build a random test database, or just about anything else that makes the other team members' lives easier.

I spent a large chunk of time on one project writing a form handler that let the other developers arrange labels and text boxes on forms. On another project I built a tool that helped developers define complex menu hierarchies more easily. (The development environment we were using back then was bad at both designing forms and building menus.)

The programs written by a toolsmith are often somewhat unfinished. They may contain bugs and may require their users to follow certain paths or risk falling into untested parts of the program. Still, if the tool is useful enough, it's worth living with a few quirks.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The reason you can get away with less-than-perfect applications in-house is that future sales don't depend on the program working perfectly. Outside customers might refuse to buy later releases of the program and may flame you in online discussion groups, but your coworkers are stuck with you.

Fixing Bugs Is Dangerous

When you fix a bug, there's a chance that you'll fix it incorrectly, so your work doesn't actually help. There's also a chance that you'll introduce one or more new bugs when you fix a bug.

In fact, you're significantly more likely to add a bug to a program when you're fixing a code than when you're writing new code from scratch. When you write new code, you (hopefully) understand what you want the code to do and how it should work. When you're fixing code sometime later, you don't have the same level of understanding.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

To reduce the problem, you need to study the code thoroughly to try to regain the understanding you originally had. Hopefully, you were paying attention in Chapter 7, "Development," when I said you should write code for people not the computer. If you did, then it will be much easier for you to figure out what the broken piece of code is trying to do.

Finally, whether you fix the bug correctly, other pieces of code may rely on the buggy behavior. When you change the code, you may break other pieces of code that were (at least apparently) working before.

Which Bugs to Fix

There may be some good reasons not to fix every bug, but in general bugs are bad, so you should remove as many of them as possible. So how do you decide which bugs to fix and which to put in the "fix later" category?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

To decide which bugs you should fix, you should use a simple cost/benefit analysis to prioritize them. For each bug, you should evaluate the following factors.

- **Severity**—How painful is the bug for the users? How much work, time, money, or other resources are lost?
- **Work-arounds**—Are there work-arounds?
- **Frequency**—How often does the bug occur?
- **Difficulty**—How hard would it be to fix the bug? (Of course, this is just a guess.)
- **Riskiness**—How risky would it be to fix the bug? If the bug is in particularly complex code, fixing it may introduce new bugs.

After you evaluate all the bugs, you can assign them priorities. Note that you may want the priorities to change over time. If your next release is a long time away, you can focus on the most severe bugs without work-arounds. If your time is limited, you can focus on the least risky bugs so that you don't break anything else before the next release.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

LEVELS OF TESTING

Bugs are easiest to fix if you catch them as soon as possible. After a bug has been in the code for a while, you forget how the code is supposed to work. That means you'll need to spend extra time studying the code so that you don't break anything. The longer the bug has been around, the greater the chances are that other pieces of code rely on the buggy behavior, so the longer you wait the more things you may have to fix.

In order to catch bugs as soon as possible, you can use several levels of testing. These range from tightly focused unit testing that examines the smallest possible pieces of code, to system and acceptance testing that exercises the system as a whole.

Unit Testing

A *unit test* verifies the correctness of a specific piece of code. As soon as you finish writing a piece of code, you should test it. Test it as thoroughly as possible because it will get harder to fix later.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

LITTLE TESTS AND BIG TESTS

It's much easier to test a lot of little pieces of code rather than one big piece. Combining many pieces of code can lead to a combinatorial explosion of the number of paths through the code that you need to test.

For example, suppose you have a piece of code that performs 10 `if-then` tests. Depending on a set of values, the code takes one branch or another at each of the 10 decision points.

To follow every possible path through the code, you need to test every possible combination of branches. For 10 branches with 2 paths each, that's $2^{10} = 1{,}024$ possibilities. If you don't check them all, you may have a combination that doesn't work.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Even if you do check them all, there's a chance that there's a bug in one of them, and you just got unlucky and didn't notice it. You should still at least touch every possible path. (If you don't walk down a path in a forest, you won't notice the snake sitting in the middle of it.)

Now suppose you break the big piece of code into 10 pieces, each containing a single `if-then` test. If you test the pieces separately, you need only two test cases for each, making a total of 20 test cases.

This is a somewhat simplistic example, but breaking big chunks of code into simpler pieces makes them much easier to test and debug.

Usually unit tests apply to methods. You write a method and then test it. If you can, you may even want to test parts of methods. That lets you catch bugs minutes or even seconds after they hatch, while they're still weak and easy to kill.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

If you're using an object-oriented programming language, be sure to test code that doesn't act like a normal method. For example, be sure to test constructors (which execute when you make a new object), destructors (which execute when you destroy an object), and property accessors (which execute when the program gets or sets a property's value).

Because unit tests are your first chance to catch bugs, they're extremely important. Unfortunately, it's also easy for programmers to assume the code they just wrote works. After all, if it didn't work, you would have fixed it!

Chapter 7 said that you can write more effective validation code if you write it before you write the routine it protects. The same applies here. You can often do a better job on a method's unit tests if you write them before you write the method. That way you won't know what assumptions the code makes, so you won't make the same assumptions when you write the tests.

You may also want to add more test cases after you write the code, so you can look for situations that the code might not handle correctly. The "Testing Techniques" section discusses some of the kinds of tests you may want to write.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Typically, a test is another piece of code that invokes the code you are trying to test and then validates the result. For example, suppose you're writing a method that organizes Pokémon card decks. It groups the cards by evolution chain (cards that are related) and then sorts the chains by their total smart ratings (average of attack, defense, special attack, and special defense). A unit test might generate a deck containing 100 random cards, pass them into the method, and validate the sorted result. The test method could repeat the random deck test a few hundred times to make sure the sorting method works for different random decks.

Other tests may perform user actions such as opening forms, clicking on buttons, or clicking and dragging on a window to see what happens.

After you write the tests and use them to verify that your new code works, you should save the test code for later use. Sometimes, you may want to incorporate some or all the unit tests in regression testing (described in the next section).

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You also need the tests again if you discover a bug in this code. You may think the unit tests are enough to flush out every bug so there won't be any in the future, but that's not the case. Bugs eventually appear. If you've saved all your unit tests, you won't need to write them again for the routine that went wrong. You also won't need to rewrite them if there's no bug but you need to modify the code.

Of course, writing a bunch of tests can clutter up your artistically formatted code. Depending on your programming environment, you may avoid that by moving the test code into separate modules. You can also use conditional compilation to avoid compiling the test code in release builds so it doesn't make the final executable bigger.

Integration Testing

After you write a chunk of code and use unit tests to verify that it works (or seems to work), it's time to integrate it into the existing codebase. An *integration test* verifies that the new method works and plays well with others. It checks that existing code calls the new method correctly, and that the new method can call other methods correctly.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Integration typically focuses on the new code and other pieces of code that interact with it, but it should also spend some time verifying that the new code didn't mess up anything that seems unrelated.

For example, suppose you're building a program to help you design duct tape projects (things like duct tape wallets, flowers, suits of armor, and prom dresses). You just wrote a new method to build parts lists giving the amount and kinds of duct tape you need for a particular project.

The new method passes its unit tests with flying colors, so you integrate it into the existing codebase. In integration tests, the main program can call the method successfully, and the new method can call existing code to do things like fetch duct tape roll lengths and prices.

Everything seems fine until you try to use the program to order new duct online. Suddenly that part of the program is no longer working. That duct tape ordering module may seem completely unrelated to the parts list method, but somehow it's not.

For example, the parts list code might open the pricing database and accidentally leave a price record locked. You might not notice this during unit testing and integration testing, but the tape ordering part of the program won't work if that record is locked.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

To discover this kind of bug, you use regression testing. In ***regression testing***, you test the program's entire functionality to see if anything changed when you added new code to the project. These tests look for ways the program may have "regressed" to a less useful version with missing or damaged features.

Ideally, when you finish unit testing a piece of code, you would then perform integration testing to make sure it fits in where it should and that it didn't break anything obvious. Then you perform regression testing to see if it broke something non-obvious.

Unfortunately, performing regression testing on a large project can take a lot of time, so developers often postpone regression testing until a significant amount of code has been added or modified. Then they run the regression tests. Of course, at that point there may be a lot of bugs and it may be hard to figure out which change caused which bug. Some of the "new" code may also not be all that new, so some of the bugs may be a bit older and therefore harder to fix.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

To fix bugs as quickly as possible, you need to perform regression testing as often as possible.

Automated Testing

You might not have time to run through every test every day. After all, you need time to do other things like write new code, perform code reviews, and eat donuts at staff meetings. However, a good automated testing system may do it for you. Automated testing tools let you define tests and the results they should produce. Some of them let you record and replay keyboard events and mouse movements so that a test can interact with your program's user interface.

After running a test, the testing tool can compare the results it got with expected results. Some tools can even compare images to see if a result is correct.

For example, to test a drawing program, you might record your actions as you draw, resize, and color a polygon. Later the testing tool would repeat the steps you took and see if the resulting picture matched the one you got when you did it interactively.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Some testing tools can run ***load tests*** that simulate a lot of users all running simultaneously to measure performance. For example, load tests can tell if too many users trying to access the same database will cause problems in your final release.

A good testing tool should let you schedule tests so that you can run regression testing every night after the developers all go home. (Or you could start the tests running before you leave for the night.)

When you come in the next morning, you can check the tests to see if there are any newly discovered problems that you should fix before you begin writing new code.

OUTSOURCING TESTS

One annoying feature of outsourcing is that there's no good time for the clients and suppliers to meet. If it's during the middle of the work day here, it's the middle of the night there or vice versa. (I suppose that wouldn't be a problem if you're working in San Jose and outsourcing to Los Angeles, but I don't think that's typical.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

A friend of mine used the time difference to his advantage. His software development team would write code during the day. They would perform their unit and integration tests, and then ship their code to testers in India before they left for the day.

When the Indian testers arrived in the morning, the new code would be waiting for them. They would run the regression tests and send the results back to the developers, who would see them first thing the next day. (Or maybe first thing on the same day. It depends on how you look at time zones.)

The result was a lot like what you would get from an automated testing tool, but humans have a lot more flexibility than automated tools, so they can follow much more complicated instructions.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Component Interface Testing

Component interface testing studies the interactions between components. This is a bit like regression testing in the sense that both examine the application as a whole to look for trouble, but component interface testing focuses on component interactions.

A common strategy for component interface testing is to think of the interactions between components as one component sending a message (a request or a response) to another. You can then make each component record its interactions (plus a timestamp) in a file. To test the component interfaces, you exercise the system and then review the timeline of recorded events to see if everything makes sense.

THE BIG PICTURE

I've done some work with a company that processes photographs taken at popular tourist destinations such as amusement parks and sporting events. The photographers take your picture and upload it to a local computer.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Components in the application perform several processing steps such as moving the picture into a (huge) database, turning the pictures right side up (sometimes the photographers hold their cameras sideways), and creating smaller thumbnail images.

For debugging purposes, the components can write messages with timestamps into a log file so that you can see what's going on and so that you can tell if one of the components it stuck. This company processes tens of thousands of photographs every day, so if a process gets stuck for a few hours, hundreds or thousands of customers can't buy pictures of themselves standing beside their favorite theme park mascots.

Because the company processes so many images, the log files can grow extremely quickly. Depending on the level of information recorded, the logs can grow by megabytes per hour.

continues

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

(continued)

To prevent the log files from eventually gobbling up all the disk space on the planet, the components were written so that they check systemwide settings when they start to decide how much information to record. If something's going wrong, you change the settings and restart a component. (To make that possible, they're also good at picking up where they left off when you restart them.) After a few minutes, you check the log files to see what's wrong and you fix the problem. When you're done, you change the settings back to allow little or no logging, and restart again.

The ability to quickly change the amount of information recorded without recompiling has been extremely useful in keeping the process running smoothly.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Planning ahead of time for component interface testing can also help with the application's design. Thinking in terms of loggable messages passed between components helps keep the components decoupled and gives them a clearer separation. That makes them easier to implement and test separately.

System Testing

As you may guess from its name, ***system testing*** is an end-to-end run-through of the whole system. Ideally, a system test exercises every part of the system to discover as many bugs as possible.

A thorough system test may need to explore many possible paths of interaction with the application. Unfortunately, even simple programs usually contain a practically unlimited number of possible paths of interaction.

For example, suppose you're writing a program to keep track of dirt characteristics for hikaru dorodango (dirt polishing) enthusiasts, things like color, amount available, and grain size. Also suppose the program includes only a login screen and a single form that uses a grid to display dirt information. Then you would need to try each of the following operations:

- Start the program and click Cancel on the login screen.
- Start the program, enter invalid login, click OK, verify that you get an error message, and finally click Cancel to close the login screen.
- Start the program, enter invalid login, click OK, verify that you get an error message, enter valid login information, and click OK. Verify that you can log in.
- Log in, view saved information, and close the program. Log in again and verify that the information is unchanged.
- Log in, add new dirt information, and close the program. Login in again and verify that the information was saved.
- Log in, edit some dirt information, and close the program. Login in again and verify that the changes were saved.
- Log in, delete a dirt information entry, and close the program. Login in again and verify that the changes were saved.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You need all those tests for just two screens, neither of which can do much. (Even then, I've seen a lot of applications where those tests wouldn't be good enough. For example, some programs won't save changes in a grid control unless you move the cursor to another cell after changing a cell's data.)

For more complicated applications, the number of combinations can be enormous. In the end, you'll probably have to test the most common and most important scenarios, and leave some combinations untested.

Acceptance Testing

The goal of *acceptance testing* is to determine whether the finished application meets the customers' requirements. Normally, a user or other customer representative sits down with the application and runs through all the user cases you identified during the requirements gathering phase to make sure everything works as advertised.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Remember that the requirements may have changed after the requirements phase. In that case, you obviously verify that the application satisfies the revised requirements.

Acceptance testing is usually straightforward; although, depending on the number of use cases, it can take a long time. A fairly simple application might need only a few use cases. (The hikaru dorodango example described in the preceding section might need only a few to check that you can log in, view, add, edit, and delete data.) A large, complex application with detailed needs might have dozens or hundreds of use cases. In that case it might take days or even weeks to go through them all.

One mistake developers sometimes make is waiting until the application is finished before starting acceptance testing. You do need to perform acceptance testing then, but if that's the first time the customer sees the application, there may be problems. Customers may decide that their interpretation of a use case is different from yours. Or they may decide that what they need is different from what they thought they needed during requirements gathering.

In those cases, you're much better off if you do a quick run-through of each use case as soon as the application can handle it. Then if you need to change the requirements, you can do it while there's still some time left in the development schedule and not at the end of the project when all of the programmers have scheduled overseas vacations.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Other Testing Categories

Unit test, integration test, component interface test, and system test categorize tests based on their scale with unit test being at the smallest scale and system test including the entire application. An acceptance test differs from a system test in the point of view of the tester: A system tester is typically a developer, whereas an acceptance tester is a customer representative.

The following list summarizes other categories of testing that differ in their scope, focus, or point of view.

- **Accessibility test**—Tests the application for accessibility by those with visual, hearing, or other impairments.
- **Alpha test**—First round testing by selected customers or independent testers. Alpha tests usually uncover lots of bugs and defects, so they generally aren't open to a huge number of users because that might ruin your reputation for building good software.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- **Beta test**—Second round testing after alpha test. Generally, you shouldn't give users beta versions until the application is quite solid or you might damage your reputation for building good software. Sometimes, beta tests are used as a sneaky form of a limited trial to build excitement for a new release in the user community.
- **Compatibility test**—Focuses on compatibility with different environments such as computers running older operating system versions. Also checks compatibility with older versions of the application's files, databases, and other saved data.
- **Destructive test**—Makes the application fail so that you can study its behavior when the worst happens. (Obviously, if you have good backups, you won't actually destroy the code. You'll destroy the application's performance.)
- **Functional test**—Deals with features the application provides. These are generally listed in the requirements.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- **Installation test**—Makes sure you can successfully install the system on a fresh computer.
- **Internationalization test**—Tests the application on computers localized for different parts of the world. This should be carried out by people who are natives of the locales.
- **Nonfunctional test**—Studies application characteristics that aren't related to specific functions the users will perform. For example, these tests might check performance under a heavy user load, with limited memory, or with missing network connections. These often identify minimal requirements.
- **Performance test**—Studies the application's performance under various conditions such as normal usage, heavy user load, limited resources (such as disk space), and time of day. Records metrics such as the number of records processed per hour under different conditions.
- **Security test**—Studies the application's security. This includes security of the login process, communications, and data.
- **Usability test**—Determines whether the user interface is intuitive and easy to use.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

TESTING TECHNIQUES

The previous sections described some different levels of testing (unit, integration, component, system, and acceptance) and alluded to some methods for testing (try out every combination of actions that you can think of), but they didn't explain specific techniques for performing actual tests.

In particular, they didn't discuss generating data for tests. For example, suppose a method organizes Pokémon card decks as described earlier. You can test it by generating a random deck and seeing if the method organizes it correctly, but how do you know it will work with *every* possible deck?

The following sections describe some approaches to designing tests to find as many bugs as possible.

Exhaustive Testing

In some cases, you may be able to test a method with every possible input. For example, suppose you write a tic-tac-toe (noughts-and-crosses) program and one method is in charge of picking the

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

best move from a current board position. You could test the method by passing it a board position, seeing what move it picks, and then verifying that there are no better moves that it could have chosen instead.

There are only 9! = 362,880 possible board arrangements, so you could pass the method every possible combination of moves to see what it does. (In fact, many of the board arrangements are impossible. For example, you can't have three Os on the top row and three Xs on the middle row in the same game. That means there are fewer than 9! possible arrangements to test.)

This sort of exhaustive testing conclusively proves that a method works correctly under all circumstances, so it's the best you can possibly do. Unfortunately, most methods take too many combinations of input parameters for you to exhaustively try them all.

For a ridiculously simple example where an exhaustive test is impossible, suppose you write a `Maximum` method that compares two 32-bit integers and returns the one that's larger. Each of the two inputs can take roughly 4.3×10^9 values (between –2,147,483,648 and 2,147,483,647), so there are approximately 1.8×10^{19} possible combinations. Even if you had a computer that could call the method and verify its results 1 billion times per second, it would take more than 570 years to check every combination.

Because most methods take too many possible inputs, exhaustive testing won't work most of the time. In those cases, you need to turn to one of the following methods.

Black-Box Testing

In *black-box testing*, you pretend the method is a black box that you can't peek inside. You know what it is supposed to do, but you have no idea how it works. You then throw all sorts of inputs at the method to see what it does.

You can start black-box testing by sending it a bunch of random inputs. Remember that you need to perform these tests only occasionally, not every time the program runs, so you can test a *lot* of random values. For example, you might throw a few million random pairs of values at the `Maximum` method described in the previous section. It doesn't matter if it takes the test a few minutes to finish.

Even if you don't know how the method works, you can try to guess values that might mess it up. Typically, those involve special values like 0 for numbers and blank for strings. They may also include the largest and smallest possible values. For strings that might mean a string that's all blanks or all ~ characters.

Sometimes, you can trip up a method that expects to process names by using strings containing numbers or special characters such as &#%!$ (which looks like a cartoon character swearing).

Some methods don't work well if their inputs include a lot of duplicates, so try that. For example, quicksort is one of the fastest sorting algorithms usually, but it gives terrible performance if the items it is sorting all have the same value. (Consult an algorithms book or search for quicksort online if you want to see the details.)

If a method takes a variable number of inputs, make sure it can handle 0 inputs and a really large number of inputs. If it takes an array or list as a parameter, see what it does if the array or list is empty or missing.

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Finally, look at boundary values. If a method expects a floating point parameter between 0.0 and 1.0, make sure it can handle those two values.

White-Box Testing

In *white-box testing*, you get to know how the method does its work. You then use your extra knowledge to design tests to try to make the method crash and burn.

White-box testing has the advantage that you know how the method works, so you can try to pick particularly difficult test cases. Unfortunately it has the disadvantage that you know how the method works, so you might skip some test cases that you assume work.

For example, you might know that a method would be confused by zero-length strings. But you knew that when you wrote the code, so you handled it. The problem is, you may not have handled it correctly. If you handled everything correctly, then there wouldn't be any bugs and you wouldn't need testing at all.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Use white-box testing to create tests you know will be troublesome, but don't skip tests that you "know" the method can handle.

Gray-Box Testing

Gray-box testing is a combination of white-box and black-box testing. Here you know some but not all the internals of the method you are testing. Your partial knowledge of the method lets you design specific tests to attack it.

For example, suppose a method examines test score data to find students that might need extra tutoring help. You don't know all the details, but you do know that it uses the quicksort algorithm to sort the students by their grades. In that case, you might want to see what the method does if every student has the same grade because that might mess up quicksort. (Because you don't know what else is going on inside the method, you also need to write a bunch of black-box styles tests.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

BLACK-BOX AND WHITE-BOX TESTING

With black-box testing, if you truly don't know how a method works, then it's harder to assume it handles specific cases correctly. Unfortunately with black-box testing, you don't know where to look for weaknesses.

White-box testing lets you specifically attack a method's weaknesses, but as mentioned a couple of times (both in this chapter and in Chapter 7) it's easy for programmers to assume their code works. (That's the biggest drawback of white-box testing.) That can make them skip some test cases that might uncover a bug.

You can get the best of both worlds by combining black-box and white-box testing. One way to do that is to have two different people test a method. The programmer who wrote it can build some white-box tests, and someone else can design some black-box tests.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

> Many software projects have designated testers who do nothing but try their hardest to destroy their colleagues' code. Sometimes their attitude can be a bit adversarial, but the results can be remarkable if team members don't take things too seriously. (There's a great short article about IBM's Black Team at `www.t3.org/tangledwebs/07/tw0706.html`.)
>
> Another approach that can give some of the same benefits is to have developers write black-box tests before they write a method's code. (Okay, these might really be more like "dark-gray-box" tests because a developer might have some idea about how he will write the method. You can probably do even better if you have one person write the black-box tests and then have another write the code.) Then after the method is written, its author can create white-box tests to go with it.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

TESTING HABITS

Just as there are good programming habits, there are also good testing habits. These habits make testing more effective so you're more likely to find bugs quickly and relatively painlessly. They also make it less likely that new bugs will appear when you fix a bug.

The following sections describe some testing habits that can make you a better tester.

Test and Debug When Alert

In Chapter 7, I said that you should write code when you're most alert. That helps you understand the code better so that it reduces the chances of you writing incorrect code and adding bugs to the application.

Similarly, you should test and debug when you're alert. Then when something goes wrong, you'll be more likely to understand what the program is supposed to be doing, what it is actually doing, and how to fix it. Debugging while tired is a good way to add new bugs to the program.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

(DWT stands for "driving while texting" and is illegal in most U.S. states. DWT can also stand for "debugging while tired," and it should be illegal, too.)

One nice thing about automated tests is that they don't get tired. You may be exhausted after a long day of coding, but a testing tool can exercise the application while you catch up on your sleep. Then in the morning you can start refreshed chasing any bugs that were found.

Test Your Own Code

Before you check your code in and claim it's ready for prime time, test it yourself. This is the last chance you have to find your own bugs before someone else does. Save yourself some embarrassment and do your own work. If you make someone else do it for you, they may decide to rub your nose in it for days or weeks to come.

Stories abound that tell of programmers who don't test their code before checking it into the project. One of my friends who was a project manager hung a toy skunk outside the door of the developer

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

who broke the weekly build. It stayed there until someone else broke a build. One time the skunk stayed outside my friend's door for more than a month, so no one was immune to the skunk. (That sort of thing can be amusing, but only if everyone takes it with good humor. Some people couldn't handle that sort of thing.)

Another group I heard of had a programmer whose name happened to be Fred. Pretty much every week Fred managed to break the build, so the other programmers would spend several hours "de-fredding" the code.

Lots of larger projects have that "one guy" who messes up the project build. Don't be that guy.

Have Someone Else Test Your Code

It's important to test your own code, but you're too close to your code to be objective. You have assumptions about how it works that unconsciously influence the tests you perform. To find as many bugs as possible, you also need someone with a fresh perspective to test it.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Even if you're Super Programmer (faster than a speeding binary search, more powerful than a linked list, and able to leap tall b-trees with a single bound), you're going to make mistakes every now and then. You've spent a lot of time and effort on your code, so when someone gently points out your mistakes, it's easy to become defensive. Your feelings are hurt. You feel personally attacked. You pull into yourself like a spurned teenager and start playing emo music on your earphones. (In the worst case, you retreat into your fortress of solitude and become a super-villain.)

In fact, all that actually happened is that someone else found a mistake that anyone could have made. They didn't cause the mistake; it was already sitting there waiting to pounce during a demo for the company president. (And I've seen that happen! A lot!) You should be grateful that the bug was caught before it escaped into a released product where it could embarrass your whole programming team.

Mistakes happen all the time, particularly in software development. It's important to thank the tester for pointing out this flaw, fix it, and move on with no hard feelings.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The ability to take this kind of criticism can be such an important factor in software engineering that Gerald Weinberg coined the term "egoless programming" in his book *The Psychology of Computer Programming*. Even though he wrote that book way back in 1971, the term is still important in programming today. (The latest edition of his book is *The Psychology of Computer Programming: Silver Anniversary Edition*, Dorset House, 1998).

THE RULES OF EGOLESS PROGRAMMING

Here's a summary of Gerald Weinberg's Ten Commandments of Egoless Programming:

1. **Understand and accept that you will make mistakes.** Everyone makes mistakes. (Even me after 30+ years of programming experience.) Try to avoid mistakes, but realize that they will occur anyway. No one else programs without any mistakes, so why should you?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

2. **You are not your code.** Just because you wrote a piece of flawed code, that doesn't make you a bad person. Don't take the bug home with you and ruin your weekend obsessing over it. Be glad the bug was found when it was. (And wish it had been found sooner!)

3. **No matter how much "karate" you know, someone else will always know more.** Even the greatest programmers of all time sometimes learn from others. And chances are, some of the people around you have more experience, at least in some facets of programming. Learn what Yoda has to offer.

4. **Don't rewrite code without consultation.** By all means fix bugs, but don't rewrite sections of code without consulting with your team. Bulk rewrites should be performed only for good reasons (like replacing a buggy section of code or rearranging code so that it can be broken up into separate methods), not because you don't like someone's indentation or variable names. If it ain't broke, don't fix it.

5. **Treat people who know less than you with respect, deference, and patience.** Even you started out as a programming novice. You made simple mistakes, did things the hard way because you didn't know better, and asked naive questions (if you were smart enough to ask questions). Be patient and don't reinforce the stereotype that good programmers are all prima donnas. (Also see #3. You may know more than someone, but not everyone.)

   One of the lessons I've learned over the years is that good ideas sometimes lie behind bad code. A piece of code that you think is weird may be trying to address an issue that isn't obvious. Stay humble and find out what the programmer was trying to do. Then decide if there's a better way to deal with the issue.

6. **The only constant in the world is change.** After a while, programmers tend to become comfortable with what they know. Unfortunately, change happens anyway whether you like it or not. Embrace change and see if it can work in your favor.

   (I worked on one project with about 25 programmers and around 100,000 lines of object-oriented code. Unfortunately the project manager said flat out that he "didn't get object-oriented code." He learned to program before object-oriented languages were invented and he didn't see the point. That made him practically useless in any technical discussion.)

   At the same time, don't discard something just because something new has come along. Like programmers, techniques that stand the test of time do so because they're useful.

7. **The only true authority stems from knowledge, not from position.** Don't use your position (as lead developer, senior architect, or even corporate vice president) to force your point of view down others' throats. Base your decisions on facts and let the facts speak for you.

continues

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

(continued)

8. **Fight for what you believe, but gracefully accept defeat.** Programming tasks rarely have a single unambiguous solution. There's *always* more than one way to tackle a problem. If the group doesn't decide to take your approach, don't worry about it. If the result is good enough, then it's good enough.

   Later if it turns out you were right and the approach taken wasn't good enough, don't rub it in. That attitude makes it harder for the group to make good decisions in the future. (Besides, some day you'll be on the wrong side of a decision and your coworkers will be slow to forget the time you acted all high and mighty.)

9. **Don't be "the guy in the room."** Sometimes you may need to close your office door and bang out some code, but don't go into hibernation and emerge only briefly to restock your Twinkie and NOQ energy drink supply. Stay engaged with the other developers so you can collaborate with them effectively.

10. **Critique code instead of people—be kind to the coder, not to the code.** This can be as simple as a subtle wording change. Instead of saying, "What were you thinking you utter moron?" you could say, "It looks like this variable isn't being initialized before it's passed into this routine." Okay, that example is a bit extreme, but you get the idea. Make comments that refer to what the code is doing not to the person who wrote it.

    Comments should also be positive if possible and focus on improving the code instead of dwelling on pointing out what's wrong. Instead of, "This variable isn't being initialized," you could say, "We should probably initialize this variable." (Notice how that comment also treats the code as group property instead of one person's mistake? It's good to help developers think of it as a joint project and not a collection of code owned by specific people.)

Fix Your Own Bugs

When you fix a bug, it's important to understand the code as completely as possible. If you wrote a piece of code, you probably have a greater understanding of it than your fellow programmers do. That makes you the logical person to fix it. Anyone else will need to spend more time coming up to speed on what the code is supposed to do and how it works.

If someone else fixes your code and does it wrong, your code looks bad. It may not be your fault (well, ultimately it was because you made the initial mistake), but you're the one who gets credit for the new bug. You may end up having to fix your own problem and the new one.

Besides, if you made a mistake, it may be useful to fix it yourself so that you can learn how to avoid that mistake in the future.

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Think Before You Change

It's common to see beginning programmers randomly changing code around hoping one of the changes will make a bug go away. (Sadly, you sometimes also see those sorts of random changes in experienced programmers.)

I won't say this is the *worst* way to debug code but only because I'm sure someone out there can come up with an even more terrible method. However, this is certainly an extremely bad way to fix software. If you're making random changes, you're not paying attention to what the changes are doing. If a change makes a bug disappear, you don't really know if it fixed the bug or just hid it. You don't know if the change added a new bug (or several). You also missed out on an opportunity to learn something so you won't make the same mistake in the future.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

CARD COUNTING

In college I had a roommate whose professor made everyone work with punched cards. (If you don't know what those are, see `en.wikipedia.org/wiki/Punched_card`.) It took several minutes to an hour to get the Computer Center to run a deck of cards, and the professor's theory was that using cards instead of typing code into the computer interactively would discourage people from trying to fix a program by trial-and-error.

Of course, what students did was make four or five copies of their decks (which could contain several hundred cards each) so they could make four or five random changes per session. It just goes to show how clever people can be at being stupid.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Don't Believe in Magic

Suppose you've spent hours chasing a bug. You've made some test changes and the bug has gone away. It's remarkable how many developers stop at that point, pat themselves on the back, and call it a job well done.

Unless you know why the changes you made fixed the bug, you can't assume the bug is really gone. Sometimes you've just hidden it. Or perhaps it went away for completely unrelated reasons, like your order processing center in New York just shut down for the evening and stopped sending you new orders.

Before you cross a bug off of your To Do list, make sure you understand exactly what changes you made and why they worked. (Also ask yourself if the changes will have bad consequences.)

See What Changed

If you're debugging new code, you can't check an older version to see what changed, but if you're chasing a bug in code that has been recently modified (perhaps due to a bug fix), see what's changed. Sometimes the difference makes the bug pop out and saves you hours of work.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Fix Bugs, Not Symptoms

Sometimes developers focus so closely on the code that they don't see the bigger picture. They find a line of code that contains a bug and fix it without considering whether there's a larger issue.

For example, suppose you're writing a method that calculates registration prices for a bull riding competition. Unfortunately, the method is giving senior citizen discounts to people who don't deserve them. (People over 85 get $3 off, but the program is giving them to younger contestants, too.)

You step through the code for a few problem customers, and you discover the bug is in the following calculation:

```
age = current_year - birth_year
```

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

It turns out some people have entered their birthdates in the format mm/dd/yy. For example, assuming it's 2015, someone born in 2005 who enters her age as 4/1/05 will have a calculated age of 2015 – 05 = 2010. With an age of more than 2,000 years, she's certainly old enough for the discount.

One way to fix this would be to check the customer's birthdate and, if the year doesn't contain four digits, not offer the discount. You might anger a few 104-year-olds, but at least you won't have parents accusing you of encouraging 12-year-olds to ride bulls.

This fix works (sort of), but it doesn't address the real problem: Customers are entering their birthdates in the wrong format. A better solution would be to modify the user interface to require customers to enter their birthdates in the required format. (You could also add some assertions to look for a valid format to make sure this sort of bug doesn't reappear later.)

Look at the entire context of the code that contains a bug and ask yourself whether you're fixing a bug or a symptom of something bigger. Make sure you understand the whole problem before you act.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Test Your Tests

If you write a bunch of tests for a method and those tests don't find any bugs, how do you know they're working? Perhaps the tests are flawed and they don't detect errors correctly.

After you write your tests, add a few bugs to the code you're testing and make sure the tests catch them. (Basically you need to test the tests.)

HOW TO FIX A BUG

Obviously, when you fix a bug you need to modify the code, but there are a few other actions you should also take.

First, ask yourself how you could prevent a similar bug in the future. What techniques could you use in your code? What tests could you run to detect the bug sooner?

Second, ask yourself if a similar bug could be lurking somewhere else. You just went to a lot of trouble isolating this bug. If other pieces of code contain a similar problem, it will be easier if you

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

find them now instead of waiting for them to break something else. Do a search of the rest of the project's code to see if you can find this bug's cousins.

Third, look for bugs hidden behind this one. Sometimes, the symptoms of one bug mask the symptoms of another. For example, suppose you write a method that flags customers who have unpaid balances greater than $50.00. You write a second method that sends e-mails to those customers to nag them. Unfortunately, a missing decimal point in the first method makes it find customers with balances greater than $5,000. Because you don't have any customers with such large balances, you never discover that the second method is sending e-mails to the wrong addresses. (This is sort of like asking a mechanic to fix your car's starter when you don't realize the engine is also missing.)

Fourth, examine the code's method and look for other possibly unrelated bugs. Bugs tend to travel in swarms. A piece of code may be extra complicated, poorly organized, or cluttered with badly conceived patches to previous bugs. Whatever the reason, some pieces of code are just buggier than others. When you fix a bug, look around for others. If you find a nest of bugs, ask whether you should refactor it to make it more maintainable.

Finally, make sure your fix doesn't introduce a new bug. The chances of a line of modified code containing a bug are much higher than those for an original line of code. (That combined with the fact that bugs tend to swarm means some piece of code can actually sprout bugs faster than you can fix them. It's like playing a particularly annoying game of whack-a-mole.) Take extra care to try to not cause more problems than you solve. Then thoroughly test your changes to make sure they worked and that they didn't break anything.

ESTIMATING NUMBER OF BUGS

One of the unfortunate facts about bugs is that you can never tell when they're all gone. As Edsger W. Dijkstra put it, "Testing shows the presence, not the absence of bugs." You can run tests as long as you like, but you can never be sure you've found every bug.

Similarly you can't know the number of bugs lurking in a project. (If you could, then you could just keep testing until that number reached zero.) Fortunately, there are some techniques you can use to estimate the number of bugs remaining in a program. They have some serious drawbacks, but at least they're better than nothing. (They also give you some actual, if not necessarily verifiable, numbers to report at management presentations to prove that you're doing something useful now that programming is winding down.)

Tracking Bugs Found

One method for estimating bugs is to track the number of bugs found over time. Typically, when testing gets started in a serious way, this number increases. After the testers have uncovered the most obvious bugs, the number levels off. Hopefully, the number of bugs found eventually declines. If you plot the number of bugs found per day, the graph should look more or less like the one in Figure 8-1.

When you're working out near the "getting close to zero" part of the graph, you have some reason to believe that you've found most of the bugs.

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

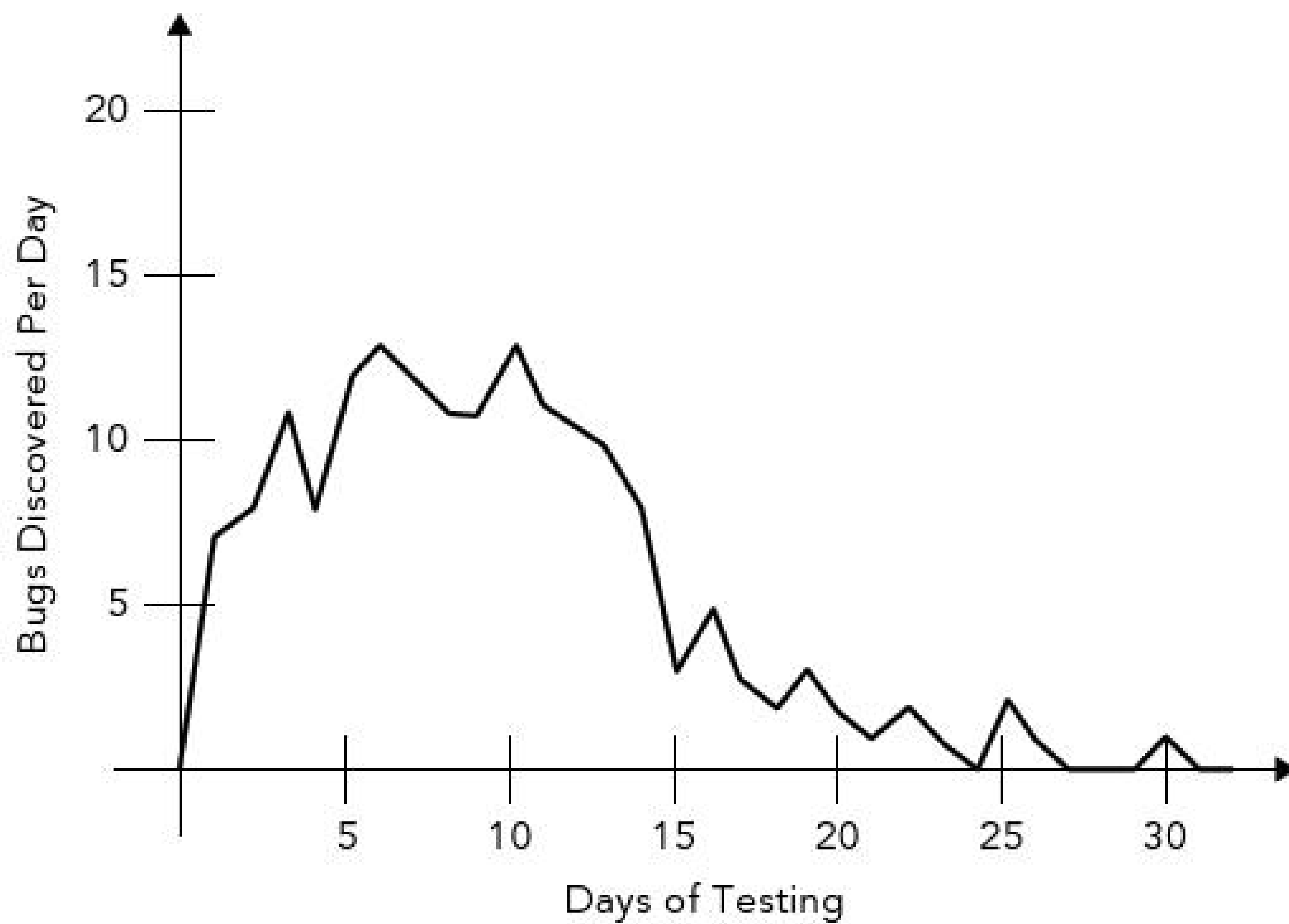


FIGURE 8-1: When you're in the "getting close to zero" part of the graph, you may be running out of bugs.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

This approach is easy, intuitive, and doesn't require a lot of extra work (beyond finding the bugs, which you need to do anyway), so it's a good start. (Graphs are also good in management presentations. You can make them colorful and people can pretend to understand them.)

Unfortunately, this approach has a couple of problems. First, it tends to track the easiest bugs to find. After 4 weeks of testing, you may have found 80 percent of the easy bugs but only 5 percent of the tricky bugs. The graph declines because you're running out of easy-to-find bugs, but there may still be plenty of sneakier bugs lying in wait.

Similarly, this kind of estimate assumes your test coverage is equally good on all parts of the project. If you've neglected part of the application or failed to look for a particular kind of bug (for example, invalid customer data), there may be a whole slew of bugs remaining that you don't know about. Sometimes, you can see this effect when you add a new test to your automated test suite and suddenly a whole bunch of new bugs appear.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

CODE COVERAGE

Some testing tools can measure *code coverage*, the lines of code that are executed during a demonstration or a suite of tests. They can tell you how many times a particular piece of code has been exercised.

You should use code coverage tools to make sure that every part of the system is visited at least once by the tests. Executing a line of code doesn't guarantee that you've found any bug in that line. However, if you don't execute a chunk of code, you're guaranteed not to find any bugs hiding there.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Seeding

Another approach for estimating bugs is to "seed" the code with bugs. Simply scatter some bugs throughout the application.

Run your tests and see how many of the artificial bugs you find. If the unintentional bugs are about as good at hiding as the bugs you planted, you should be able to estimate the number of bugs remaining.

For example, suppose you insert 40 bugs in the code and your tests find 34 of them. That 85-percent success rate implies that you may have found 85 percent of the real bugs. If you've found 135 real bugs so far, then there may have originally been approximately 135 ÷ 0.85 ≈ 159 bugs. That means there are about 159 – 135 = 24 bugs remaining.

The previous approach (tracking found bugs) assumes the bugs you've found are representative of the bugs as a whole. The seeding approach makes a similar assumption. It assumes the artificial bugs can accurately represent the true bugs.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Unfortunately, it's a lot easier to create simple bugs by tweaking a line of code here and there than it is to create complex bugs that involve interactions between several methods in different modules. That means the seeding method can greatly underestimate the number of complicated and subtle bugs.

The Lincoln Index

Consider the following word problem.

> *Suppose you have two testers Lisa and Ramon. After they bash away at the application for a while, Lisa finds 15 bugs and Ramon finds 13. Of the bugs, they find 5 in common. In total, how many bugs does the application contain?*

The correct answer in this case is, "Wait, I thought this was a software engineering book, not a mathematics text. You didn't say I was going to have to solve word problems!"

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Of course you don't really know how many bugs are in the application, but the Lincoln index gives you a guess. In this example, the Lincoln index is 15 × 13 ÷ 5 = 39.

More generally, if two testers find E_1 and E_2 errors respectively, of which S are in common, then the Lincoln index is given by the following equation:

$$L = \frac{E_1 \times E_2}{S}$$

Like all the other bug estimation techniques, this one isn't perfect. It relies on the assumption that the testers have an equal chance to find any particular bug, and that's probably not true. Both testers are most likely to find the easiest bugs, so the value S is probably larger than it would be if finding bugs was completely random. That means the Lincoln index probably underestimates the true number of bugs.

Another way the Lincoln index can break down is if Lisa and Ramon have similar testing styles. In that case, their common style may tend to lead them to find the same bugs. Again the value S would be larger than it would if bugs were found randomly, and the Lincoln index would be smaller than it should be.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

NOTE *The Lincoln index was described by Frederick Charles Lincoln in 1930, long before the invention of modern computers. He was an ornithologist who used the method to estimate the number of birds in a given area based on the number of birds counted by different observers. For more information about the Lincoln index, see* `en.wikipedia.org/wiki/Lincoln_index`.

HOW DOES THE LINCOLN INDEX WORK?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Suppose the two testers have probabilities P_1 and P_2 of finding any given bug and assume the application contains B bugs. Then you would expect them to find $E_1 = P_1 \times B$ and $E_2 = P_2 \times B$ bugs, respectively.

The chance of a particular bug being found by ***both*** testers would be $P_1 \times P_2$, so you would expect them to find $S = P_1 \times P_2 \times B$ bugs in common.

Plugging those values into the formula for the Lincoln index gives

$E_1 =$

$$L = \frac{E_1 \times E_2}{S}$$

$$= \frac{(P_1 \times B)\ (P_2 \times B)}{P_1 \times P_2 \times B}$$

When you get through canceling, all that's left is B. That means you should expect the Lincoln index to be about the same as B, the total number of bugs.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

SUMMARY

If all programs contain bugs, you may be tempted to throw your hands up in the air, walk away from your software engineering job, and open a bakery. Even though you generally cannot remove every bug from a program, you can usually remove enough bugs that the remaining ones don't appear too often and don't inconvenience users too much.

The key to finding bugs so that you can remove them is testing. By constantly testing code at small, medium, and large scales, you can find bugs as soon as possible and make removing them easier. Continue testing until bug estimation techniques indicate that you may have caught most of the important bugs.

When your testing efforts aren't finding much to fix, it's time to start deployment. The next chapter describes typical deployment tasks and some of the things you should do to make deployment easier.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

EXERCISES

1. Two integers are *relatively prime* (or *coprime*) if they have no common factors other than 1. For example, 21 = 3 × 7 and 35 = 5 × 7 are *not* relatively prime because they are both divisible by 7. By definition –1 and 1 are relatively prime to every integer, and they are the only numbers relatively prime to 0.

   Suppose you've written an efficient `IsRelativelyPrime` method that takes two integers between –1 million and 1 million as parameters and returns `true` if they are relatively prime. Use either your favorite programming language or pseudocode (English that sort of looks like code) to write a method that tests the `IsRelativelyPrime` method. (Hint: You may find it useful to write another method that also tests two integers to see if they are relatively prime.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

2. What changes do you need to make to the `IsRelativelyPrime` method to test all the testing code? In other words, what do you need to do to test the testing code?

3. What testing techniques did you use to write the test method in Exercise 1? (Exhaustive, black-box, white-box, or gray-box?) Which ones *could* you use and under what circumstances?

4. What limitations do the tests you wrote for Exercise 1 have? Would a particular testing technique help?

5. The following code shows a C# version of the `AreRelativelyPrime` method and the `GCD` method it calls.

```
// Return true if a and b are relatively prime.
private bool AreRelativelyPrime(int a, int b)
{
    // Only 1 and -1 are relatively prime to 0.
    if (a == 0) return ((b == 1) || (b == -1));
    if (b == 0) return ((a == 1) || (a == -1));

    int gcd = GCD(a, b);
    return ((gcd == 1) || (gcd == -1));
}

// Use Euclid's algorithm to calculate the
// greatest common divisor (GCD) of two numbers.
// See http://en.wikipedia.org/wiki/Euclidean_algorithm
private int GCD(int a, int b)
{
    a = Math.Abs(a);
    b = Math.Abs(b);

    // If a or b is 0, return the other value.
    if (a == 0) return b;
    if (b == 0) return a;

    for (; ; )
    {
```

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary
9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

```
        int remainder = a % b;
        if (remainder == 0) return b;
        a = b;
        b = remainder;
    };
}
```

The `AreRelativelyPrime` method checks whether either value is 0. Only –1 and 1 are relatively prime to 0, so if `a` or `b` is 0, the method returns `true` only if the other value is –1 or 1.

The code then calls the `GCD` method to get the greatest common divisor of `a` and `b`. If the greatest common divisor is –1 or 1, the values are relatively prime, so the method returns `true`. Otherwise, the method returns `false`.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Now that you know how the method works, implement it and your testing code in your favorite programming language. Did you find any bugs in your initial version of the method or in the testing code? Did you get any benefit from the testing code?

6. Write an exhaustive test for the `AreRelativelyPrime` method in pseudocode. What are the benefits and drawbacks to this version?

7. Write a version of the program you wrote for Exercise 5 that uses an exhaustive test. How large can you make the range of values (to the nearest powers of 10) and still finish testing in under 10 seconds? Approximately how long would it take to test with the range –1 million to 1 million?

8. Does all this this seem like a lot of work?

9. Exhaustive testing actually falls into one of the categories black-box, white-box, or gray-box. Which one is it and why?

10. The section "The Lincoln Index" describes an application where Lisa found 15 bugs, Ramon found 13 bugs, and they found 5 in common. The Lincoln index estimates that the application might contain approximately 39 bugs in total. After you fix all of the bugs that Lisa and Ramon found, how many are left?

11. Suppose you have three testers: Alice, Bob, and Carmen. You assign numbers to the bugs so the testers find the sets of bugs {1, 2, 3, 4, 5}, {2, 5, 6, 7}, and {1, 2, 8, 9, 10}. How can you use the Lincoln index to estimate the total number of bugs? How many bugs are still at large?

12. What happens to the Lincoln estimate if the two testers don't find any bugs in common? What does it mean? Can you get a "lower bound" estimate of the number of bugs?

13. What happens to the Lincoln estimate if the two testers find *only* bugs in common? What does it mean?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

14. The Lincoln index has a statistical bias, so some people prefer to use the Seber estimator:

$$Bugs = \frac{(E_1 + 1) \times (E_2 + 1)}{(S + 1)} - 1$$

Repeat Exercise 10 with the Seber estimator. How does it compare to the Lincoln index estimate?

15. Suppose two testers find 7 and 5 bugs respectively but none in common. Repeat Exercise 12 with the Seber estimator.

16. Suppose two testers find only the same bugs. Repeat Exercise 13 with the Seber estimator.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

WHAT YOU LEARNED IN THIS CHAPTER

- Goals of testing
- Reasons not to remove a bug (diminishing returns, deadlines, it's too soon since the last release, the bug is useful, the code will soon be obsolete, it's a feature not a bug, at some point you need to release something, the program is already worth using, fixing bugs is dangerous)
- How to decide which bugs to fix (severity, work-arounds, frequency, difficulty, riskiness)
- Levels of testing (unit, integration, component interface, system, acceptance)
- Uses for automated testing
- Testing categories (accessibility, alpha, beta, compatibility, destructive, functional, installation, internationalization, non-functional, performance, security, usability)
- Testing techniques (exhaustive, black-box, white-box, gray-box)
- Good testing habits (test when alert, test your own code, have someone else test your code, use egoless programming, fix your own bugs, think before you change, don't believe in magic, see what changed, fix bugs not symptoms)
- How to fix a bug (How can you prevent similar bugs in the future? Could the bug be elsewhere? Look for bugs hidden by this bug. Look for unrelated bugs. Make sure your fix doesn't introduce another bug.)
- Methods for estimating number of bugs (tracking, seeding, Lincoln index, Seber estimator)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary
9b3c108192e5fcb5ac110e4bbde32ac1
ebrary
9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9 Deployment

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Plans are nothing; planning is everything.

—Dwight D. Eisenhower

WHAT YOU WILL LEARN IN THIS CHAPTER:

- What you should put in a deployment plan
- Why you need a rollback plan
- Cutover strategies
- Common deployment tasks
- Common deployment mistakes

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

After you've built the next blockbuster first-person shooter, financial projection tool, or Goat Simulator, it's time for deployment. ***Deployment*** is the process of putting the finished application in the users' hands and basking in their adulation.

At least in theory. In reality deployment can be a nightmare unrivaled by any step in the software engineering process. It can be the stage when you discover that the program that worked perfectly in testing scenarios is a total failure in the real world. It can be the point when you realize that all your months or years of labor slaving over an overclocked CPU has been for naught. It can be when you and your coworkers learn how many resumes per hour the laser printer down the hall can produce.

Fortunately, the reality usually falls somewhere between the user adulation and nightmare scenarios. Most things work, more or less, with a few notable exceptions that give you interesting stories to tell later at the wrap party. (In the words of Captain Jack Sparrow in *Pirates of the Caribbean: Dead Man's Chest*, "Complications arose, ensued, were overcome.")

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

IMPLEMENTATION AND INSTALLATION

In addition to the term "deployment," some people use the terms *implementation, installation*, and *release*. In this context, they all mean basically the same thing; although they may show slight differences in the speaker's background.

Programmers tend to think of "implementation" as writing code to do something. (As in, "Did you implement the user validation module yet?") "Installation" sounds more humdrum than the other terms. You call an electrician or a plumber to install something. Besides, "deployment" sounds more dynamic. You don't "install" or "implement" troops into a field of battle.

Software developers do occasionally talk about releasing programs into the wild.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

This chapter describes the deployment phase of a software engineering project. It explains deployment scope and lists some of the things you should consider in a deployment plan.

SCOPE

A project's scope can range from a small tool you wrote for your own use, to in-house business software that will be used by hundreds or thousands of users. Some of the largest projects (things like operating systems, browsers, and game console games) might have millions of users.

In addition to the number of users, scope includes the size of the application. It includes the amount of data involved, the number of external systems that are affected, and the sheer quantity of code (all of which could fail).

As you can probably guess, larger deployments provide more opportunities for mistakes. Big projects have more pieces that can go wrong. They also provide more combinations of little things that can add up to big problems.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

For those reasons, small deployments are usually the smoothest. If you write an application for your own use and it doesn't work, you've only inconvenienced yourself and you have no one else to blame. If you roll out a new version of an operating system to millions of customers and then immediately discover you need to send out a security fix, you lose credibility. (Yes, that scenario happens all the time.)

Before you begin deployment planning, you should consider the scope of the deployment and plan accordingly. How much pain will failure during deployment cause? How much of that pain will come back to haunt you? If a failure will inconvenience a lot of users, or make the users unable to help their customers, you should spend extra time writing the best deployment plan possible. The next section explains in general what you need to put into a deployment plan.

THE PLAN

If everything went according to plan, you could write down a simple list of steps to follow and then work through them with guaranteed success. Unfortunately, the real world rarely works that way. Something always goes wrong. Perhaps not everything, but something.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

When the inevitable emergency occurs, how well you recover depends largely on how thoroughly you planned for unexpected situations. If you have a backup plan ready to go, you may work around the problem and keep moving forward. If you don't have a backup plan, you may need to stop the deployment and try again later.

Even stopping a deployment can be difficult and dangerous if you don't plan for it. After you drive over a cliff, it's a little late to say, "Oh wait. I forgot something. Let's try this tomorrow."

To start deployment planning, list the steps that you hope to follow. Describe each step in detail as it is supposed to work.

Next, for every step, list the ways that step could fail. Then describe the actions that you will take if one of those failures occurs. Describe work-arounds or alternative approaches that you could use.

This part of planning can be extremely hard. It's not always easy to think of work-arounds for every possible disaster. Sometimes it may not even be possible.

For example, suppose your application requires 40 new networked computers with 8 GB of memory. What will you do if the network doesn't work? Or if the computers don't arrive from the manufacturer? Or if the manufacturer sends you eight computers with 40 GB of memory? In those cases, you may be unable to continue the deployment in any meaningful way. You may think your software installation is bulletproof, but hardware issues bring deployment to a screeching halt before it gets started. In that case, the "solution" to those problems might be to delay the deployment and fix the problems (or die trying).

For a slightly less obvious example, suppose your computers arrive on schedule and they work just fine, but there's something wrong with the network and you're not getting the bandwidth you should so the users can only process four or five jobs per day instead of the normal 15 to 20. You *could* move the users onto the new system anyway, but that would cause unnecessary pain and suffering (and you'll get your fair share). At this point, it would be better to postpone the deployment for a day or two, figure out what's wrong with the network, and start over.

After you've worked through all the plan's desired steps and anticipated as many problems as possible, write a rollback plan that lets you undo everything you've done. Be sure you can restore any other applications that you've updated and any data that you've converted for the new system.

Unfortunately, rolling back some of those sorts of changes can be difficult. For example, suppose your new application will run on a new operating system. If something goes wrong, restoring the older operating system can be a huge pain.

There are a few things you can do to make such a major restoration possible. For example, you can make complete images of the computers you're updating so that you can put them back exactly as you found them if necessary.

At some point, the pain of retreat is greater than the pain of moving forward. Some call that the *point of no return.* People often underestimate how painful moving forward can be, however, so it's good to delay the point of no return as much as possible. It's one thing to say, "We'll just press onward and let the users deal with any problems that crop up." It's another thing entirely to face management when the database fails and the users are reduced to writing customer orders on pieces of paper.

Often the action that determines the point of no return is moving users to the new system. You can set up networks, install new printers, and spray your company logo on new computers, but until

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

people are using the new application, it's relatively easy to go back. The next section discusses the process of moving people to the new application.

CUTOVER

Cutover is the process of moving users to the new application. There are several ways you can manage cutover. For some applications, you can just post the new version on the Internet and let users grab it. For other projects, you may be able to e-mail a new version to users, or you may be able to just install the new system on users' computers.

More interesting deployments require that you do a bunch of set up (upgrading operating systems, converting data into new formats, and installing coffee machines) before you can move users to the new system.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

During the setup time, the users may be unable to do their jobs. To minimize disruption, it's important that the whole process go as smoothly as possible. The following sections describe four ways you can make life easier for all concerned: staged deployment, gradual cutover, incremental deployment, and parallel testing.

Staged Deployment

If you can't reduce the impact of catastrophic failures, you can sometimes reduce their likelihood by using staged deployment. In *staged deployment*, you build a *staging area*, a fully functional environment where you can practice deployment until you've worked out all the kinks.

After you have the installation working smoothly, you can test the new application in an environment that's more realistic than the one used by the developers. You can use the staging area to find and fix a few final bugs before you inflict them on the users.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

If you can, use power users to help do the testing. They'll know what problems the other users are most likely to encounter. (Users can also break a system in ways no programmer or tester can.) Staged testing will also give them a preview of what's coming. Hopefully, they'll like what they see and tell the other users how wonderful their future lives will soon become.

When you're fairly certain that everything is ready for prime time, you sneak in at night and perform the actual deployment on the user's computers, like Santa leaving presents in children's stockings. Hopefully you leave presents and not a lump of coal. (You don't really have to sneak in at night, but many companies do basically that. They have IT personnel upgrade the users' computers at night or over the weekend to minimize disruption.)

You still need a deployment plan in case something unexpected goes wrong. Just because everything works flawlessly in the staging environment doesn't mean it will on the users' machines. However, staging should have reduced the number of major problems you encounter.

Gradual Cutover

In *gradual cutover*, you install the new application for some users while other users continue working with their existing system. You move one user to the new application and thoroughly test it.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

When you're sure everything is working well, you move a second user to the new system. When that user is up and running, you install a third user, then a fourth, and so on until everyone is running the new application.

The advantage to this approach is that you don't destroy every user's productivity if something goes wrong. The first few guinea pigs may suffer a bit, but the others will continue with business as usual until you work out any tangles in the installation procedure. Hopefully, you'll stumble across most of the unexpected problems with the first couple of users, and deployment will be effortless for most of the others.

One big drawback to this approach is that the system is schizophrenic during deployment. Some users are using one system while others are doing something different. Depending on the application, that can be hard to manage. You may need to write extra tools to keep the two groups logically separated, or you may need to impose temporary rules of operation on the users.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

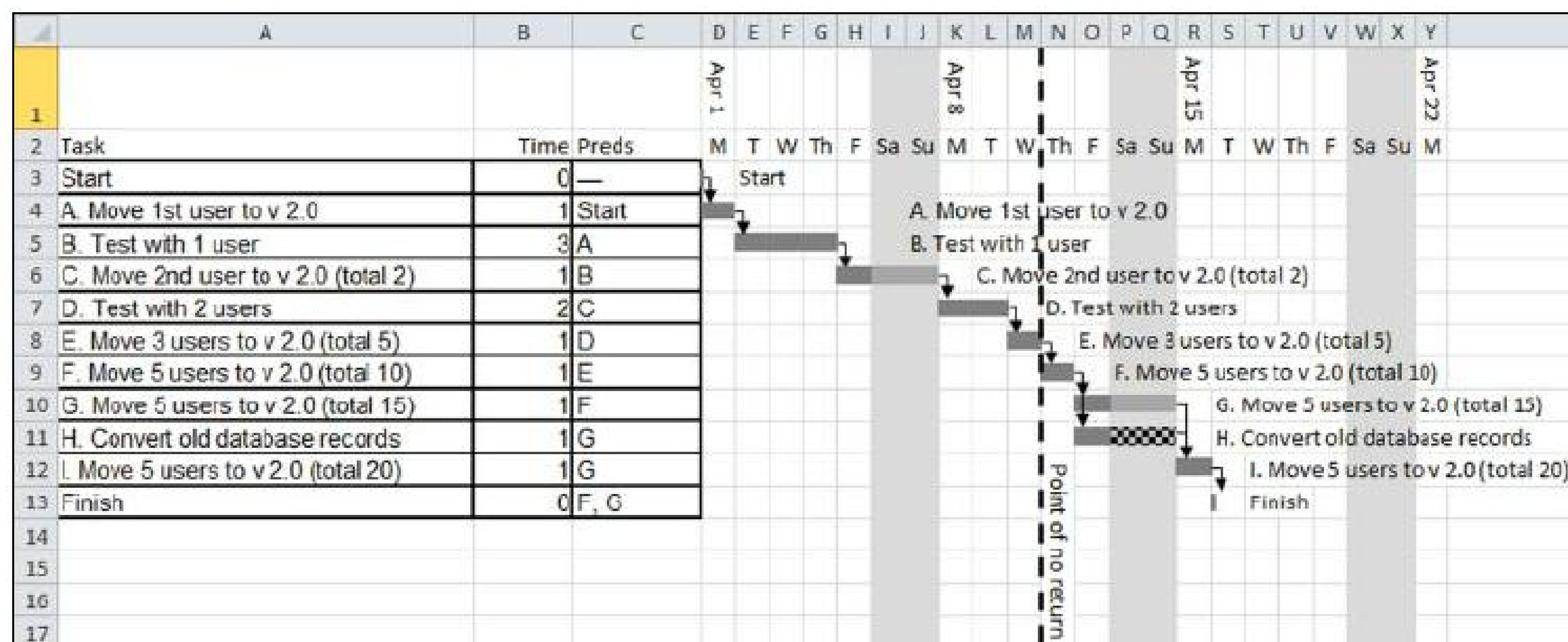
For example, suppose you're building version 2.0 of your AdventureTrek program, an application that lets customers make reservations for adventure treks such as BASE jumping off of national monuments, kayaking over waterfalls, and hang gliding over active volcanos. Unfortunately, the new version uses an updated database format to accommodate your latest offering: wing-walking on jets.

Now consider what happens when you move a user to the new system. The database is full of records in the old format. Either the 2.0 user must work with the old records, or the system must route the old records to users that are still on version 1.0. After the 2.0 user creates some new records, the system must route those records only to that user because the others can't read a version 2.0 record.

Eventually you'll move all the users to the new version and, at that point, no one will work with the older records. Obviously you need to convert the older records into the new format at some time. Of course, once you do, people using version 1.0 won't be able to do anything, so you'll need to switch them all over to version 2.0 right away.

Figure 9-1 shows a Gantt chart that gives one possible schedule for migrating all 20 users to the new version.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary



Task	Time	Preds
Start	0	—
A. Move 1st user to v 2.0	1	Start
B. Test with 1 user	3	A
C. Move 2nd user to v 2.0 (total 2)	1	B
D. Test with 2 users	2	C
E. Move 3 users to v 2.0 (total 5)	1	D
F. Move 5 users to v 2.0 (total 10)	1	E
G. Move 5 users to v 2.0 (total 15)	1	F
H. Convert old database records	1	G
I. Move 5 users to v 2.0 (total 20)	1	G
Finish	0	F, G

FIGURE 9-1: This schedule takes 11 work days to migrate all 20 users to AdventureTrek 2.0.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The schedule starts by moving one user to the new version. Pick one of the power users for this so that user can help exercise the new version thoroughly. The schedule then calls for three days of testing with this user on the new version.

Next, the schedule moves a second user to version 2.0. Testing continues for two more days with just two users on the new version.

If everything is going smoothly after this point, the schedule starts moving groups of users to the new version. It moves three more users to the new version, making a total of five. The next day it moves five more users, and it moves five again on the following day.

At this point (Friday, April 12), 15 users are using version 2.0, five users are using version 1.0, and the database contains a mix of old and new record formats.

If you move the last users to the new version, then no one will be able to work with the older records, so it's time to convert the database.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Depending on the volume of business in the application, you may need to convert the database before cutover is finished. For example, if the old records require a *lot* of maintenance, then five users on the old version may not be enough. In that case, you might want to convert the data after 10 users are on the version 2.0.

Now the schedule upgrades the final five users and converts the old data in the database. Because database conversions often take longer than expected, the schedule places the conversion on a Friday, so the database developers can work over the weekend if necessary. The extra time is represented in the Gantt chart by a checkerboard pattern to indicate that it might not be necessary.

> **NOTE** *Working the occasional Friday night and weekend is the price many software developers pay for keeping the users productive, but don't abuse them. If you make developers work too many evenings and weekends, their work-related productivity will drop and their resume-polishing productivity will soar.*

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Incremental Deployment

In *incremental deployment*, you release the new system's features to the users gradually. First, you install one tool (possibly using staged deployment or gradual cutover to ease the pain). After the users are used to the new tool, you give them the next tool.

This method doesn't work well with large monolithic applications because you usually can't install just part of such a system. (Imagine building a new air traffic control system and installing only the part that lets planes take off. You'd have to program really fast to get the landing parts of the application in place before anyone runs out of fuel.)

This method often works nicely with the iterated development approaches described in Chapters 13 and 14. There programmers build one feature at a time and, when a feature is ready, it's released to the users.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Parallel Testing

Depending on how complicated the new system is, you might want to run in parallel for a while to shake the bugs out. For example, if you have enough users, you could have a handful of them start using the new system in parallel with the old one. They would use the new system to do their jobs just as if the new system were fully deployed.

Meanwhile another set of users would continue using the old system. The old system is the one that actually counts. The new one is used only to see what would happen if it were already installed.

After a few days, weeks, or however long it takes to give you enough confidence in the new system, you start migrating the other users to the new system. You can ease the process by using staged deployment and gradual cutover if you like.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

DEPLOYMENT TASKS

The tasks you need to perform for a successful deployment depend on the application you're installing. A simple program like FileZilla (a really nice, free FTP program) just installs a new version of itself and you're ready to go. If you're building a customer support center from scratch, you've got a lot more work to do.

The following list itemizes some of the things you might need to deal with for a large deployment.

- **Physical environment**—These are physical things that the users need such as cubicles or offices, desks, chairs, power, lighting, telephones (possibly including headsets), and motivational posters (such as waterfalls, soaring eagles, and cats hanging from clotheslines). Plus everything that goes into any work environment such as restrooms, coffee machines, and supply closets (where employees can steal staples and rubber cement).
- **Hardware**—This includes network hardware (such as cables, fiber, switches, routers, and gateways), printers, scanners, CD or DVD burners, backup hardware, disk farms, database hardware, external hard drives, call routers, and, of course, the users' computers.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- **Documentation**—This can include some combination of physical and online documentation. It might include training materials, user manuals, help guides, and cheat sheets listing common commands.
- **Training**—If the application is complicated or very different from what users currently have installed, you may need to train the users. For larger installations, developers may have to train the trainers (either professional instructors or power users) who will then train the users.
- **Database**—Most nontrivial applications include some sort of database. Depending on the database, you may need to install database software on one or more central database servers and on the users' computers. You may also want extra hardware and software to provide extra data security features such as backups, shadowing, and mirroring.
- **Other people's software**—This is software that you didn't write. It includes systems that interact with your application (purchasing systems, web services, file management tools, cloud services, and printing and scanning tools) and other software that users need to be

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

productive (e-mail, chat, browsers, search engines, trouble-shooting databases, and word processors). Plus, of course, the operating system.

- **Your software**—This is the application you've built. It includes the application itself, plus any extra tools you've created. It also includes monitoring and testing tools that let you make sure the application is working correctly.

Of course, your project's needs will vary. You may not need telephone headsets and you may need extra motivational posters.

DEPLOYMENT MISTAKES

The basic steps for successful deployment are (1) make a plan, (2) anticipate mistakes, and (3) work through the plan overcoming obstacles as they arise. If something goes wrong and you don't have an easy fix, rollback whatever you've done, study the problem, and try again later. You can reduce the inconvenience for users by using staged deployment, gradual cutover, incremental deployment, and parallel testing.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

If you do a good job of following those steps, you should eventually get even the most complicated application up and running. Occasionally, however, a deployment fails so spectacularly that nothing can save it. Or the deployment finishes, but with all the fun and carnival atmosphere of a root canal.

The following list summarizes some of the easiest ways to torpedo an otherwise viable project:

- **Assume everything will work**—This may seem like a rookie's mistake, but many people assume their deployment plan will just magically work. Maybe you'll get lucky and that will be true, but you should probably assume it won't.
- **Have no rollback plan**—Rolling a deployment back can be a real hassle, but it's usually better than living with whatever damage you do during a failed deployment.
- **Allow insufficient time**—If everything goes smoothly, you won't need much time, but when something goes wrong, all bets are off. A deployment that should take hours could take days or even weeks. Allow extra time for unexpected problems. Then hedge your bets by scheduling the end of deployment on a Friday so that you can work into the weekend if the plan goes off the rails.
- **Don't know when to surrender**—It's easy to work around one or two small issues that don't play out as expected, but how do you know when to stop? If you keep pushing through (or around) little issues (and sometimes big ones), eventually all the compromises add up to give you a terrible result. (Like a beginning poker player with a pair of threes being gradually sucked into a huge pot.) Define conditions under which you'll fold and try again later. For example, you might quit after 4 hours or after three things go wrong. Or you might use a point system with 1 point for a trivial change, 2 points for a small work-around, and 5 points if you can't get something to work. When you get to 5 points, quit for the day.
- **Skip staging**—Staging can be time-consuming and expensive, particularly if you need to install new hardware and software. However, for a complicated deployment, staging is crucial. It lets you work out all the deployment glitches so that you don't need to completely trash the users' computers.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- **Install lots of updates all at once**—It's tempting to install a lot of updates at the same time so that you don't need to inconvenience the users repeatedly. Unfortunately, the more things you try to do at once, the more likely it is you'll run into problems. Limit the number of things you try to deploy all at once. Save the rest for a later deployment.
- **Use an unstable environment**—Have you ever used a computer where the scanning software works (sometimes), the print queues seem to get stuck randomly, and your video editing software sometimes won't import certain kinds of files? If the tools you use don't work together consistently, then you have other problems you should fix before you start a new deployment. Sometimes finding the right combination of tools that can work together can be challenging. Adding a new application will only make things worse.
- **Set an early point of no return**—If you explicitly set a point of no return, you don't need to figure out how to roll back any changes after that point. Unfortunately, you don't always know how bad things might get near the end of the deployment. The last installation task could be a total disaster that takes you days to figure out. You should set the point of no return as late as possible in the deployment schedule so that you can retreat whenever necessary. Even better, don't have a point of no return!

There's a common theme to these methods for failure. They all assume things will go well. Perhaps this is more of the unbounded optimism that makes programmer's fail to test their code. You just wrote the deployment plan and you didn't see anything wrong with it. If you had, you would have fixed it. The logical conclusion is that everything will work perfectly. That means you don't need a rollback plan, sufficient time, surrender conditions, staging, and a late point of no return.

Assume you will have problems. If you also assume that some of those problems may be big, you'll be ready in case you need to cancel the deployment and start over. If you prepare for the worst, the worst that will happen is you'll be pleasantly surprised when things go well.

SUMMARY

The basic strategy for successful deployment is straightforward. Make a plan that anticipates as many problems as possible, and then follow the plan. If big unexpected problems occur, roll back any changes you've made and try again later.

There are still a few details to take care of. For example, you need to know when to abandon a deployment attempt and try again another day. (He who quits and runs away, lives to deploy another day.) You can also use cutover strategies to make things easier.

As long as you make a plan and realize that some things will almost certainly go wrong, you should do okay and eventually get the application up and running. After that (and perhaps a celebratory team dinner at a nice restaurant), the application moves into maintenance. During this phase, your application serves its intended purpose (drawing electronic schematics, tracking orders, posting pictures of cats, or whatever), and the users send you comments, suggestions, change requests, and bug reports. (And once in a great while, a "thank you" that makes the whole thing seem worthwhile.)

At this point in your project, you've finished initial development. You gathered requirements, created high- and low-level designs, written tons of code, tested the code (and fixed some bugs), and deployed the application to the users. You're probably more than ready for a break. All you

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

want to do is run off to Disneyland, Aruba, or wherever you consider the happiest place on Earth. Unfortunately, there are a few things you need to take care of before you disappear in addition to arranging for a pet sitter. The next chapter describes tasks that you should perform at the end of a project before all the developers go their separate ways.

EXERCISES

1. Suppose you've written a small tool for your own use that catalogues your collection of pogs. You're planning your third upgrade and you need to revamp the database design. Which cutover strategy should you use?

2. Suppose you're writing an application that includes a lot of separate tools. One creates work orders, a second assigns jobs to employees, a third lets employees edit jobs to close them out, and so forth. Which cutover strategies could you use when deploying a new version of this application?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

3. Suppose the application described in Exercise 2 uses a database. Each of the pieces needs to use the database and you need to change the database structure for the new deployment. Does that change your answer to Exercise 2?

4. Suppose you're writing a large application with thousands of users scattered around different parts of your company. Which cutover strategy would you use?

5. Suppose you're building a new MMO (massively multi-player online game) and you expect to have tens of thousands of users. (Your business plan says within the next 18 months.) Users will download and install your program. What cutover strategy should you use?

6. President Eisenhower was big on planning. If you Google around a bit, you can find several quotes by Eisenhower extolling the virtues of planning (including the quote at the beginning of this chapter). Here's another quote from his remarks at the National Defense Executive Reserve Conference on November 14, 1957.

   > *I tell this story to illustrate the truth of the statement I heard long ago in the Army: Plans are worthless, but planning is everything. There is a very great distinction because when you are planning for an emergency you must start with this one thing: the very definition of "emergency" is that it is unexpected, therefore it is not going to happen the way you are planning.*

   If emergencies don't happen the way you're planning, then why make a plan in the first place? Does this apply to deployment plans?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

7. Suppose you just released a version 3.0 of your popular shareware program Fractal Frenzy, which lets users draw fractals, zoom in and save coordinates, make movies zooming in and out, and generally make cool pictures. Unfortunately, the day after the release, you discover a bug that prevents users from saving coordinates so that they can't return to saved pieces of a fractal. What should you do? Tell people right away? Wait until there's a fix? Wait until the next release?

8. Suppose you're the manager of the Internal Software Development department at a medical device manufacturer. One of your projects, Test Track, records quality test results for the devices your company makes. Depending on the device, testers record between a few dozen and several hundred test measurements per week. Your software lets testers perform data analysis to see whether the products are up to scratch.

   Unfortunately, you just learned about a bug that makes the product occasionally examine the wrong device's data. About once a month, for no obvious reason, a tester requests data on one device but gets results about a different device. Repeating the query once or twice seems to get the right results.

   What should you do? Should you rush out an emergency patch? Wait until the next major update? Ignore the problem and hope it will go away?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

WHAT YOU LEARNED IN THIS CHAPTER

- A project's scope influences how thoroughly a plan must anticipate every possible problem.
- A deployment plan should include the steps needed for deployment, possible places where things can go wrong, and work-arounds for them.
- You should be able to roll back any changes you make if a deployment becomes stuck.
- The point of no return is where it would be more painful to roll back a failing deployment than to press ahead. (If you have a good rollback strategy, then you don't need a point of no return.)
- Three cutover strategies are staged deployment, gradual cutover, and incremental deployment. Parallel testing can also help as a prelude to full deployment with one of the three strategies.
- Deployment tasks may include:
  - Physical environment
  - Hardware
  - Documentation
  - Training
  - Database
  - Other people's software
  - Your software
- Common mistakes during deployment include:
  - Assuming everything will work
  - Having no rollback plan
  - Allowing insufficient time
  - Not knowing when to surrender
  - Skipping staging
  - Installing a lot of updates at once
  - Using an unstable environment
  - Setting an early point of no return

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

10

Metrics

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You can't control what you can't measure.

—Tom DeMarco

Measuring programming progress by lines of code is like measuring aircraft building progress by weight.

—Bill Gates

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Grouping defects by importance or task
- Using Ishikawa diagrams to discover root causes of problems
- Defining and using attributes, metrics, and indicators
- Understanding the difference between process and project metrics
- Using size and function point normalization to compare projects with different sizes and complexities

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

At this point, you've finished the project. Congratulations! Some of your team members are probably itching to move on to whatever comes next, whether they plan to continue maintaining this project, start a new one, or leave to achieve that lifelong ambition of becoming a barista.

However, you should do a few more things before the team scatters to the four corners of the IT industry. Chief among those is a discussion of the recently completed project to determine what you can learn from your recent experiences. You need to analyze the project to see what went well, what went badly, and how you can encourage the first and discourage the second in the future. To do that, you need to find ways to measure the project. (Exactly how do you measure the project's "wonderfulness?")

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

This chapter describes tasks that you should perform after initial development is over. It discusses methods you can use to analyze defects (which include change requests, bugs, and other vermin) so that you can try to anticipate and minimize similar defects in the future. It also explains metrics that you can use to measure the project's characteristics and how you can use those metrics when you work on future projects.

METICULOUS METRICS

Like most software engineering tasks, gathering metrics doesn't happen only in one place (in this case, at the end of the project). It's easier to gather metrics throughout the project rather than waiting until the end. For example, you should keep track of the project's status (lines of code written, bugs fixed, milestones missed, and so forth) as you go along. I've put metrics in this chapter because at the end of a project you can look back with a new perspective and see how it all unfolded.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

WRAP PARTY

You've finished the project! That's no small feat, so you should do something as a team to celebrate. Have a party, company picnic, trip to an amusement park, or some other wrap-up activity. At least have lunch together and joke about all the times the customers altered the specifications, changed their minds about what hardware to use, and asked why you were using C++ instead of COBOL. Let the healing begin.

Note that the wrap party cannot be just another project meeting but with cupcakes, balloons, and "I Survived Project Ennui" T-shirts. Feel free to gossip about company politics, argue about whether the corporate vision statement makes sense if you read it backward, and speculate about whether upper management will be indicted for insider trading. Don't discuss outstanding bugs, analyze metrics, or turn the party into a group performance review. Do that some other time.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

It's obvious that a software organization can't succeed unless its customers are satisfied, but it also can't function unless its employees are happy. A wrap activity helps bring closure to the project and makes people feel like they accomplished something.

DEFECT ANALYSIS

At a philosophical level, any time an application doesn't do what it's supposed to, you can consider it a bug. For example, when you first start a project, it doesn't do anything. Unless that's its desired behavior, you could think of that as a bug. (If that *is* the desired behavior, let me know because I've already written that application.)

Some development methodologies actually come pretty close to that point of view.

- Task: Create a new Add Customer form.
- Bug: It doesn't let you enter a customer name.
- Change: Add a Customer label and text box.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- Bug: It doesn't let you enter a customer address.
- Change: Add an Address label and address text boxes.
- Bug: There's no OK button.
- And so forth.

However, when you're thinking about bugs with an eye toward preventing them in the future, it's helpful to differentiate among different ways the program isn't working correctly.

Kinds of Bugs

At the highest level, you can group all incorrect features into *defects*. You can then categorize defects into *bugs* (code that was written incorrectly) and *changes*. (The code is doing what the specification said to do, but the specification was wrong.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Note that it may not be anyone's fault that the specification was wrong. For example, the customers' needs may have changed since the project started. Or the environment may have changed, as when a new operating system is installed or management decides everything must move to the cloud. (Hopefully they know what the cloud is.)

The following sections describe several other ways you can categorize defects.

Discoverer

One important way to group defects is by who reported them. Bugs that are found and fixed by programmers are often invisible to the customers. The customers never need to know all the dirty little secrets that went into building the final application.

In contrast, changes that are requested by customers are obviously visible to the customers. Generally you should satisfy as many change requests as possible, as long as they don't mess up the schedule. (They give you brownie points you can spend later to resist the customers' efforts to shorten the schedule.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The worst combination is a bug that is discovered by the customers. If a bug gets to the customers, it must have snuck past code reviews, unit tests, and integration tests. They're somewhat embarrassing and reduce the customers' confidence in your team's ability to produce a high-quality application. (They also reduce your brownie points.)

For each defect, ask three questions:

- How could you have avoided the defect in the first place?
- How could you have detected the defect sooner?
- For customer-discovered defects, how could you have found the defect before the customers did?

Severity

This categorization is quite obvious. Assign a severity to each defect and focus on those that are most severe. You can use a 1 to 10 scale (or 1 to 100 scale, or whatever) if you like, but you probably don't need that level of detail. Usually, you can simply assign each defect the severity Low, Medium, or High.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Focus on the high severity defects, and for each one ask how you could have avoided it, how you could have detected it sooner, and (for customer-discovered defects) how you could have found it before the customers did. (Do these questions seem familiar?)

Time Created

You can further categorize defects by when they were created. Defects tend to snowball, so those created earlier in the project usually have greater consequences than those created later. For example, a defect added during high-level design has a lot more potential to cause pandemonium than a defect added in the last module written.

By now you can probably guess what I'm going to say next. Focus on the defects that were created earliest because they can cause the most damage. For each defect, ask how you could have avoided it, how you could have detected it sooner, and (for customer-discovered defects) how you could have found it before the customers did.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Age at Fix

Defects are like cancer: The longer they go undetected, the greater the potential consequences. Group defects by the length of time they existed before they were detected and fixed. Focus on those that remained in hiding the longest and ask the usual three questions.

Task Type

Another way to categorize defects is by the type of task you were trying to accomplish when it was created. By the type of task I don't mean "trying to write a `for` loop" or "writing a vibrant and profound sentence for the specification." I mean things like Specification, High-Level Design, User Interface Design, or Database Code.

The types of tasks you should use will depend on the project. For example, if you're writing a finance application that will run on desktop systems, then you probably don't need a Phone Interface category.

Some typical task categories include the following:

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- Specification
- Design
  - High-Level
  - Security
  - User Interface
  - External Interface
  - Database
  - Algorithm
  - Input/Output
- Programming
  - Tools
  - Security

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

  - User Interface
  - External Interface
  - Database
  - Algorithm
  - Input/Output
- Documentation
- Hardware

The previous methods for categorizing defects focus on what's most important. The errors discovered by users, have high severity, were created early, and that remained undiscovered for a long time tend to have the greatest impact, so they're important. After you identify them, you can ask the three questions to see how you can avoid the same problems in future projects.

In contrast, task categories don't identify the most important defects. Instead they try to group defects by common causes. Defects that were added while performing similar tasks may have similar causes and (hopefully) similar solutions.

For example, suppose you discover that a lot of defects originated in the specification. In that case, many of them may have a common cause such as not paying attention to the customer, not studying the user's current process enough, or unrealistic customer requests. In that case, you may be able to fix a whole bunch of defects in future projects by addressing a single issue. Perhaps if you spend a bit more time running through use cases with the customers before you finalize the specification, you can avoid some of these defects.

Ishikawa Diagrams

To figure out in which category a defect belongs, ask what task was being performed when the defect was created. For example, suppose you discover a defect on the login screen. The code incorrectly validates the user's name and password. Password validation is a security feature, so this task might fall into the Programming/Security category.

Often, however, a defect is the end of a sequence of events that was started by some primordial mistake. In this example, suppose the code does exactly what the security design said it should. In that case, the error is actually in Design/Security, not in the code.

It's also possible that this defect has an even more distant cause. Perhaps the security design correctly reflected what was described by the specification. In that case, the error is in the specification, not the design or the code.

Perhaps the specification, design, and code are all correct, and the error is in the database. Or worst of all, perhaps two or more pieces of the puzzle contain errors that combine to create the defect. (On television crime shows, a single murder leads to all the confusing clues. Imagine how much more confusing things would be if multiple crimes occurred at the same spot and muddled each other's clues.) In this example, there could be problems with any combination of the login code, the database code, the database design, the database itself, the database specification, or the security specification.

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Sometimes discovering the root the root cause of a defect can be challenging. One tool that can help is the *Ishikawa diagram* (named after Kaoru Ishikawa). These are also called *fishbone diagrams* because they look sort of like a fish skeleton. (And *Fishikawa diagrams* are an amusing blend of the two names. For your Word of the Day, look up "portmanteau" in the dictionary and ignore the definitions that deal with luggage.) They're also called *cause and effect diagrams*, but a name that prosaic won't impress anyone at IT cocktail parties.

QUALITY CONTROL

Kaoru Ishikawa used these diagrams in the late 1960s to manage quality in the Kawasaki shipyards. They've been used extensively in quality management for industrial processes.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

This is another one of those tools like PERT charts and Gantt charts (see Chapter 3, "Project Management") that are so useful for managing projects in general that they've been around far longer than software engineering has. (When the first colonists land on Tau Ceti e, they'll probably use a PERT chart to order the tasks they need to perform, a Gantt chart to schedule them, and an Ishikawa diagram to figure out why the sunscreen was left behind on the kitchen counter on Earth.)

To make an Ishikawa diagram, write the name of the defect you're trying to analyze (Incorrect Username/Password Validation) on the right of a sheet of paper. (This is the head of the fish.)

Next draw a horizontal arrow pointing to the defect name from left to right. (This is the fish's backbone.)

Now think of possible causes and contributing factors for the defect. Represent them with angled arrows leading into the spine. (These are the fish's ribs.) Label each arrow with the cause you identified.

For each of the fish's ribs, think about causes and contributing factors for that rib. Add them, again with labeled arrows. Continue adding contributing factors to each of the factors you've already listed until you run out of ideas. (I confess I haven't seen a fish with this type of skeleton. Maybe you can find them in Lake Karachay, the world's foremost duping site for radioactive waste.)

Figure 10-1 shows a sample Ishikawa diagram (although many people would omit the fishy outline).

The exact format of the diagram doesn't matter too much and there are several variations in style. The only things that are really consistent among most diagrams are

- The effect or outcome is on the right.
- There's a backbone.
- Arrows (or lines) lead from causes to intermediate causes or effects.
- Arrows (or lines) are labeled.

It doesn't matter whether you use lines or arrows, and sometimes they may point from right to left if that makes them fit in the diagram better.

Figure 10-2 shows another version of the previous diagram with a different style.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

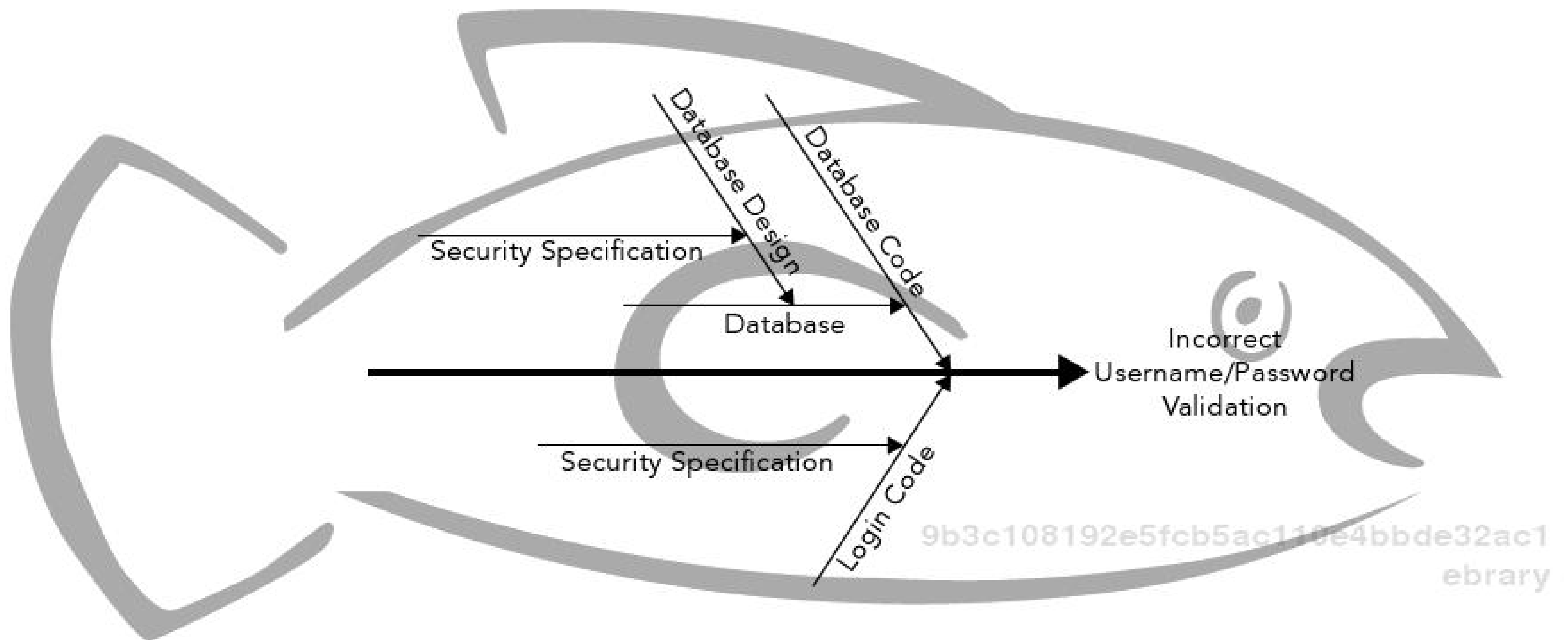


FIGURE 10-1: An Ishikawa (or fishbone) diagram shows causes leading to effects.

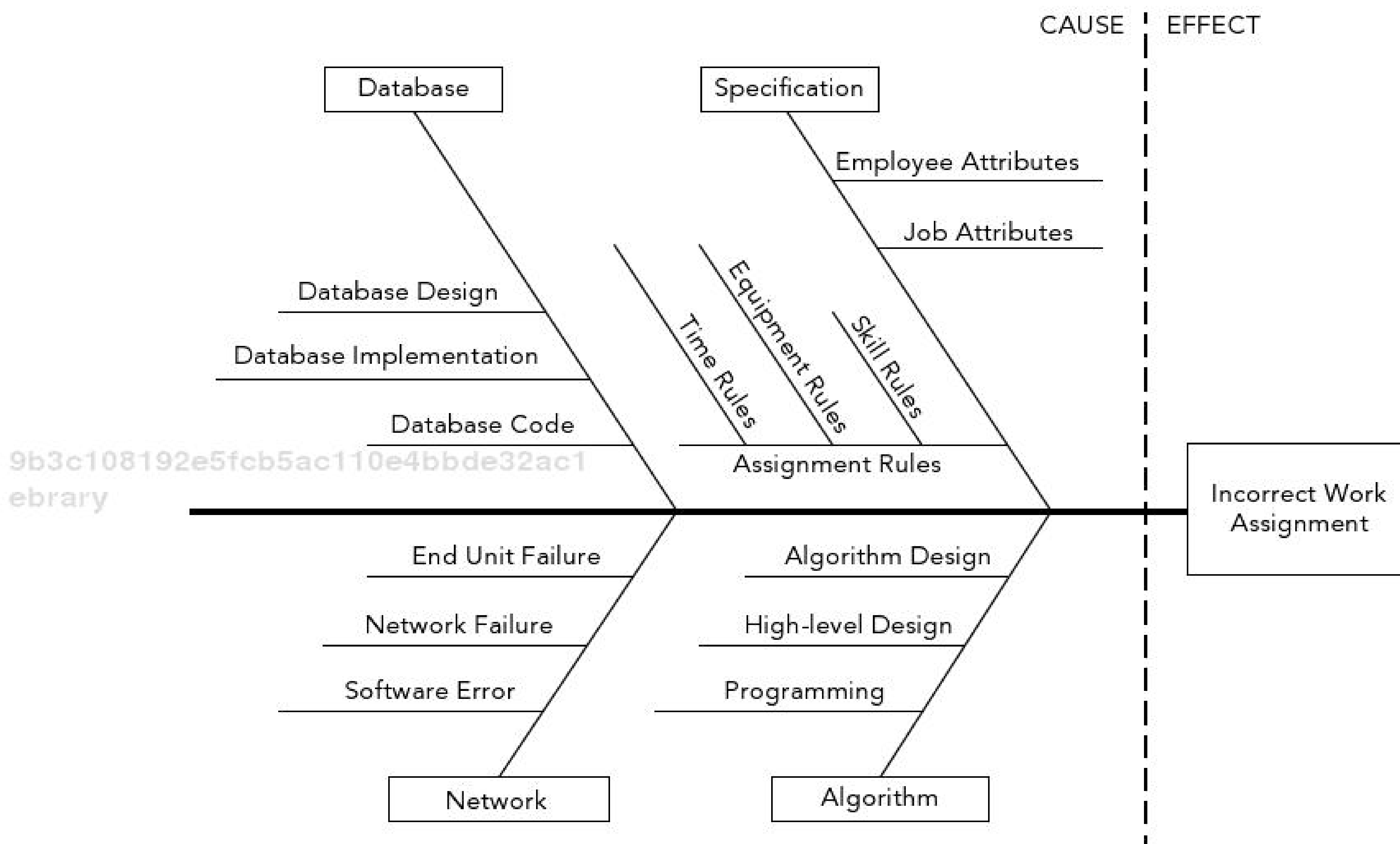


FIGURE 10-2: The exact format of an Ishikawa diagram doesn't matter as long as you can tell what causes lead to what other causes and effects.

After you build an Ishikawa diagram for a defect, take a close look at each of the possible causes and decide which ones actually helped cause the defect. Highlight causes that did play a role and cross out those that didn't. If you're not sure about a cause, study it further, possibly adding contributing causes to it.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

When you're finished, you should have discovered the root causes of the defect. You can then ask the three magic questions about the defect.

You should also use the diagram to group the defect's causes. In Figure 10-2, for example, you might find that all the problems lay in the Specification rib. In that case, you should look more closely at your specification process to see if there's something you could do to make the specification more reliable in future projects.

SOFTWARE METRICS

The defect analysis techniques described in the previous sections are more or less qualitative. They help you characterize defects based on their discoverer, severity, and age at time of removal.

In contrast, software metrics give you quantitative measurements of a project. Before you learn what kinds of metrics you can analyze, you should know a few metric-related terms.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

An ***attribute*** is something you can measure. It could be the number of lines of code, the number of defects, or the number of times the word "mess" appears in code comments.

A ***metric*** is a value that you use to study some aspect of a project. Sometimes a metric is the same as an attribute. For example, you might get useful information about a project from the number of bug reports you have received. Often metrics are calculated values. For example, you may want to look at bug reports per week or bug reports per line of code instead of just the total number of bug reports.

After you have metrics, you study them to see if any of them are good ***indicators*** of the project's future. For example, consider the metric "comments per thousand lines of code (KLOC)." If comments per KLOC is 3, that may be an indicator that the project will be hard to maintain.

You can then do two things with your indicators. First, you can use them to predict the future of your current project. For example, if you've been fixing 10 defects per week for the last 2 weeks, and you hope to clear your list of 875 defects before the initial release in just under a month, then you could be in trouble.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The second thing you can do with indicators is make strategy improvements for future projects. For example, if this project did fall into the "3 comments per KLOC" category, then you might want to change your code review process to gently encourage programmers to add a few more comments. (And to make sure they're meaningful and not just statements such as `Add 1 to num_orders`.)

SIMILAR SITUATIONS

Metrics and indicators sometimes apply only to similar projects. For example, your programmers may crank out an average of 50 lines of code per day over the course of a three-month Visual Basic desktop project. That doesn't necessarily mean they can produce the same amount of code over a two-year firmware project for a particular phone.

Metrics and indicators will be most useful for projects that are most similar. They may still be useful for other projects, but you should keep an eye on how well they are predicting a new project's future so that you can adjust your expectations if necessary.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

To summarize:

- Measure relevant attributes.
- Use the attributes to derive meaningful metrics.
- Use metrics to create indicators.
- Use indicators to predict the project's future.
- Use indicators to make process improvements.

Now that you have a little background in software metrics, the following sections give some additional details.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

COMMON COMPLAINTS

Aside from the project manager, software engineers often resist tracking metrics. Team members may feel that collecting metrics is hard and time-consuming. They may say they spend all their time measuring and counting instead of working. And besides, metrics are subjective and don't prove anything.

Some people also think metrics will be used against them to measure how productive (or unproductive) they are.

Metrics are sometimes a bit subjective and ambiguous, but any measurements are better than nothing. (Exploring a vast cavern with a book of matches isn't as good as using floodlights, but it's better than wandering in the dark bumping into walls and falling down pits.)

You should try to explain to the team members that metrics really are useful. Try to keep the extra work to a minimum and assure people that they are used to guide the project and not to determine who writes the most documentation or lines of code. It's easy to write a lot of badly written code, so punishing someone who writes less code but with higher quality doesn't make sense anyway.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The following sections explain more precisely what attributes might make good metrics, what you can use metrics for, and how you can normalize metrics so they are meaningful for projects of different sizes.

Qualities of Good Attributes and Metrics

You can measure many attributes of a software engineering project. You can measure the number of lines of code, the customers' satisfaction level, the hours the team members spent playing *Bouncing Balls*, the font used in the specification, or the team's total number of trips to the coffee pot.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Of course, some of those attributes are hard to measure (such as customer satisfaction) and others are irrelevant. The following list gives characteristics that good attributes and metrics should ideally have.

- **Simple**—The easier the attribute is to understand, the better.
- **Measureable**—To be useful, you must measure the attribute.
- **Relevant**—If an attribute doesn't lead to a useful indicator, there's no point measuring it.
- **Objective**—It's easier to get meaningful results from objective data rather than subjective opinions. The number of bugs is objective. The application's "warmth and coziness" is not.
- **Easily obtainable**—You don't want to realize the team members' fears by making them spend so much time gathering tracking data that they can't work on the actual project. Gathering attribute data should not be a huge burden.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Sometimes it's impossible to satisfy all these requirements. In particular, some important attributes can be hard to measure. For example, customer satisfaction is extremely important, but it can be hard to quantify.

For attributes such as this one, which are important but hard to measure, you may need to use indirect measurements. For example, you can send out customer satisfaction surveys and track the number of change requests you receive.

Using Metrics

Metrics have several possible uses. You can use them to

- Minimize a schedule.
- Reduce the number of defects.
- Predict the number of defects that will arise.
- Make defect removal easier and faster.
- Assess ongoing quality.
- Improve finished results.
- Improve maintenance.
- Make sure a project is on schedule.
- Detect risks such as schedule slip, excessive bugs, or features that won't work and adjust staffing and work effort to address them.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

As I mentioned in the previous section's tip, metrics and indicators work best for projects similar to those during which you gathered your metric data. Two projects that use different development methodologies, programming languages, or user environments may not always produce the same results. That means you need to use some common sense when you use indicators to try to predict a project's future.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

However, it's probably a bigger mistake to completely ignore what an indicator is telling you. Suppose in previous projects you've noticed that a low number of pages of program documentation gave you lots of bugs. Just because your current project is using a different programming language, that doesn't mean this indicator is wrong. If the programmers are producing fewer pages of documentation but the bug rate remains low, try to figure out why.

It could be that the new language is more self-documenting. (Previous projects used assembly language but this one's using Visual Basic.)

It could be you have a really good programming team on this project. In that case, you'll probably need that extra documentation for long-term maintenance when these programmers all wander off to new projects.

It could also be the case that the programmers have been working through the easy stuff first and work will become much harder later. In that case, you need to be sure the amount of documentation picks up as the difficulty level increases.

Don't ignore what your metrics are saying. If they contradict the facts, learn why so that you know whether you need to adjust the metrics or the project.

TIPS FOR USING INDICATORS

You can use indicators to provide regular feedback to the team. If it looks like some part of the project is wandering away from the practices suggested by your indicators, gently nudge the project back on course.

Don't think of indicators just as harbingers of doom. (Abandon hope all ye who stray from the required number of use cases per form.) Think of them as signposts pointing in the right direction. If you get lost, use them to guide the project back to the correct path. (It may sound like MBA doublespeak, but use them as opportunities for improvement not reasons for despair.)

Don't use metrics and indicators to appraise individuals or the team as a whole. If you yell and scream at team members because they're messing up your indicators, they'll stop giving you accurate metric data. You can suggest that someone spend more time working through use cases during requirements gathering, but if you threaten them, they're just as likely to tell you they're doing the work when they aren't.

For similar reasons, make sure people aren't hideously overworked. If team members don't have time for all their assigned tasks, they'll dump the ones they consider the lowest priority. Usually that includes tracking metrics.

Finally, don't get stuck obsessing over a single metric. If you're not spending much time on code reviews and your indicator says you should be seeing a lot of bugs, but the bugs aren't there, then perhaps this isn't a problem after all. By all means try to figure out why things are going so smoothly, but don't create a problem where one doesn't already exist.

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Metrics and indicators are often grouped into two categories depending on how you use them: process metrics and project metrics. The following sections describe those categories.

Process Metrics

Process metrics are designed to measure your organization's development process. You collect them over a long time period for many projects, and then use them to fine-tune the way you do software engineering.

For example, suppose you collect data over a series of projects and you draw the graph in Figure 10-3 showing hours of code review per KLOC versus number of bugs per KLOC.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

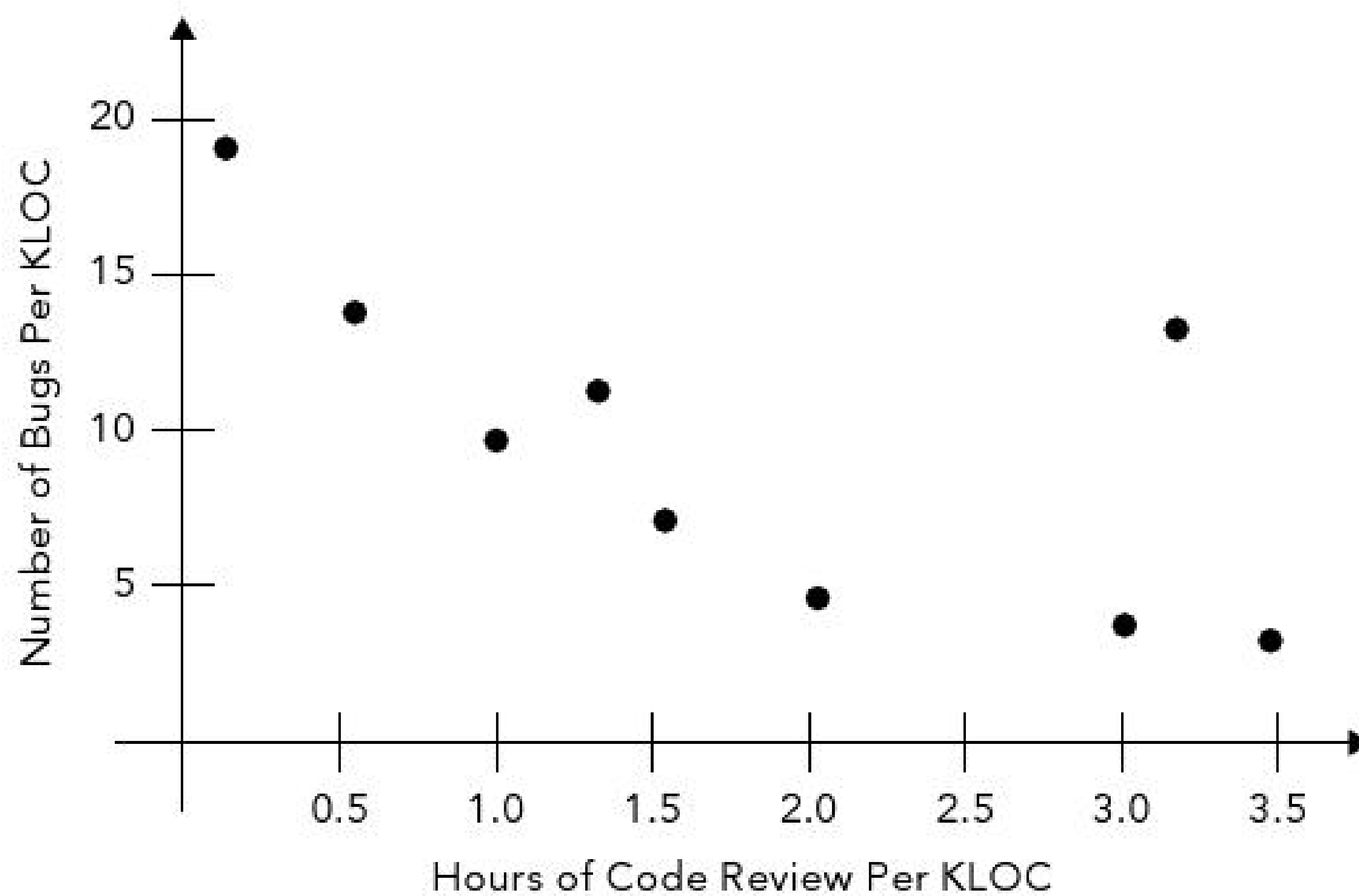


FIGURE 10-3: This graph shows the relationship between hours of code review and bugs per KLOC.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Looking at the graph in Figure 10-3, you might decide that you want to try to spend 1.5 to 2 hours of code review per KLOC in future projects. That seems to let you find most of the bugs that you would catch even if you used a lot more time on reviews. (I'd also want to dig deeper to figure out why the second project from the right had so many bugs despite the relatively large investment in code reviews. Was it a different kind of project? Was it particularly large or small? Was it run by an inexperienced technical lead?)

Project Metrics

Project metrics (which are sometimes called *product metrics* because they are about a specific software product) are intended to measure and track the current project. They let you use past performance to predict future results. Based on your predictions, you can adjust your strategy to improve those results.

You can also use project metrics to set goals. For example, suppose you have three customer representatives on your team writing use cases. Over the last week, they each managed to write an average of 10 use cases per day. You want to have 10 to 20 (call it an average of 15) use cases for the project's more complicated forms, and you have 20 complicated forms to go.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

If the numbers hold true (and they may not), you need to write 20 × 15 = 300 more use cases. At a rate of 30 use cases per day (10 for each of the three customer representatives), you should finish in about 300 ÷ 30 = 10 days. You can make that a goal: Finish writing the use cases in the next two weeks.

Things to Measure

The things you can measure on a software project are practically limitless. Fortunately, you need to track only a few metrics to get a good sense of how a project is progressing.

At a high level there are two kinds of metrics you should track: inputs and outputs. Inputs are the things you spend on the project. The following list describes some input metrics.

Cost—Money spent on the project for hardware, software, development tools, networking services, paper, training, and so forth. (For business purposes, you may also want to track salaries and overhead, but they're not as directly related to the project's performance. In contrast, if you're not spending anything on development tools, you're probably not getting the best result for your efforts.)

Effort—This is the amount of work being put into the project. It is usually measured in person-hours. Effort is relatively easy to measure.

Defect rates—The number of defects discovered over time. Defect rates are also fairly easy to measure.

Lines of code (LOC)—The number of lines of code produced per day. You might think this would be easy to measure, but it's actually kind of hard to decide what to count as a line of code. For example, do you count comments and blank lines? What about statements that are split across multiple lines? All those things pump up the line count without adding anything extra as far as the computer is concerned, but they also make the code easier to read and understand, so you should encourage programmers to use them appropriately.

Some development organizations treat a command split across multiple lines as a single line of code. Some ignore comments and blank lines. Others count blank lines up to 25 percent of the total code and ignore any blank lines over 25 percent.

It doesn't matter too much which approach you take as long as your rules are consistent across projects and the programmers don't try to game the system. (If you count comments and judge programmers on the number of lines of code they produce, you may get files with dozens of comments per actual line of code.)

Pages of documentation—There are several kinds of documentation that you might want to track. Project documentation (such as the specification and design documents) are important because they ensure that everyone is working toward a common vision. If you don't have enough of this kind of documentation, different team members may end up working at cross-purposes, resulting in extra defects and difficult long-term maintenance.

User documentation is obviously important to the end users. If you have too little, the users won't figure out how to use your program.

User documentation also reflects the complexity of the application. If you need a lot of documentation to explain the program, that may mean the design is overly complicated, and that may also indicate a lot of future defects and maintenance problems.

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You can measure all those attributes fairly directly. (At least if you can decide what to measure for LOC.) Some other attributes are harder to measure directly. They're either hard to quantify or they're subjective. The following list describes some of those items and how you might try to measure them.

- **Functionality**—How well does the application do what it is supposed to do? How well does it let the users do their jobs? This is quite subjective, but you can measure things such as the numbers of help requests, change requests, and user complaints.
- **Quality**—Do the users think of this as a high-quality application? Is it relatively bug-free? Again, this is subjective, but you can track user complaints to get some idea. You can also do user surveys. (You know, those annoying surveys that ask you how likely you are to recommend a product to your friends.)
- **Complexity**—How complex is the project? This is hard to measure directly. The amount of project documentation gives you a hint about the project's complexity. Lots of documentation may indicate a complex project that needs a lot of explaining. (Or it may just indicate you have a team member who loves to write.)

  There are a lot of other ways to estimate complexity. You can count the `if-then` statements in the code because they determine the number of paths through the code. You could also count the number of loops or other complicated code features such as recursion and particular data structures. Unfortunately, making all those counts is a fair amount of extra work.

  Function points provide another method for estimating a project's complexity. The section "Function Point Metrics" later in this chapter explains them in detail.
- **Efficiency**—How efficient is the application? In rare cases, you can calculate the theoretical maximum efficiency possible and compare the application to that. For example, you might determine that a routing program finds solutions within 15 percent of the optimal routes. In general, however, this is hard to measure.

  You can compare the users' performance to their performance before they started using your application, but you won't know if there could be an even better way to do things. (Of course, if the users were more productive before they started using your application, you might need to either write a second version or update your resume.)
- **Reliability**—How reliable is the application? This one is a little easier to measure. You can keep track of the number of times the program crashes or produces an incorrect result.
- **Maintainability**—How easy will it be to maintain the application in the long term? You can get some notion of how hard maintenance will be by looking at other metrics such as the amount and quality of the project documentation, the number of comments, and the code complexity, but usually you won't really know how maintainable the project is until you've been maintaining it for a while.

One problem with all metrics is that they're hard to apply to projects of different sizes. Studies have shown that projects of different sizes have different characteristics. For example, in larger projects team members spend more time coordinating activities than they do in smaller projects. That means they may be unable to write and debug as much code per day.

One way to make metrics a bit more meaningful for different project sizes is to *normalize* them by performing some calculation on them to account for possible differences in project size. There

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

are two general approaches for normalizing metrics: size normalization and function point normalization. The following two sections describe those two approaches.

Size Normalization

Suppose you measure the number of developers, total time, effort (in person-months), LOC, and number of bugs for projects Ruction and Fracas. Table 10-1 shows the results.

TABLE 10-1: Attributes for Projects Ruction and Fracas

	PROJECT RUCTION	PROJECT FRACAS
Developers	3	7
Time (months)	1	24
Effort (pm)	3 × 1 – 3	7 × 24 – 168
LOC	1,210	75,930
Bugs	6	462

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

In which project were the developers more productive? Which project contained buggier code? Just looking at the numbers in the table, it's hard to tell. Project Fracas includes a *lot* more code, but it also took a lot more effort. Project Ruction contained far fewer bugs, but it also contained much less code.

In *size-oriented normalization*, you divide an attribute's value by the project's size to get some sort of value per unit of size. Assuming everything about two projects is similar except for their sizes (a big assumption), the normalized metrics should be comparable.

For this example, you could divide total lines of code by the effort it took to produce the code. Similarly, you could divide the number of bugs by the total number of lines of code. Table 10-2 shows the normalized values.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

TABLE 10-2: Normalized Metrics for Projects Ructino and Fracas

	PROJECT RUCTION	PROJECT FRACAS
LOC / pm	1,210 ÷ 3 – 403	75,930 ÷ 168 – 452
Bugs / KLOC	6 ÷ 1.21 = 4.96	462 ÷ 75.93 = 6.08

The normalized values show that project Fracas was more productive in terms of lines of code for the effort (452 versus 403 LOC per person-month), but project Ruction had less buggy code (4.96 versus 6.08 bugs per KLOC).

The following list gives some of the measurements of size that you can use to normalize values.

- Number of team members
- Effort (person-months)
- KLOC or LOC

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- Cost (dollars, euros, doubloons, or whatever)
- Pages of documentation
- Number of bugs
- Number of defects
- Time (days, months, years)

Divide an attribute value by the value that makes the most sense. For example, bugs are a feature of code, so you should probably divide the number of bugs by LOC or KLOC, instead of the number of team members or effort. Similarly lines of code are produced over time by team members, so you should probably divide LOC by number of team members and number of months (which is the same as dividing by person-months).

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Other combinations, such as dividing number of bugs by the number of team members, can also have meaning hidden inside them, but they're harder to interpret.

Size-oriented metrics have a big advantage that they're usually easy to calculate. It's easy to count the number of lines of code (assuming you can agree on how to count comments and blank lines) and it's easy to count the number of person-months spent on the project, so it's easy to calculate LOC / effort. These metrics also have the advantage that a lot of project modeling applications use them as inputs.

These metrics also have a few disadvantages. One problem with normalized metrics is that you can't actually use them to predict the future unless you can already predict the future. For example, LOC / effort lets you predict how long it takes to build a project, but only if you can predict how many lines of code you need to write.

For a concrete example, suppose you know from past experience that your team can produce approximately 400 LOC / pm. If you're about to start project Rhubarb and you think it will require roughly 11,000 lines of code, then you can predict that it will take approximately 11,000 ÷ 400 = 27.5 person-months of effort. The catch is you need to know that the project will need 11,000 lines of code.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

GUESSING THE UNGUESSABLE

Although you can't know how many lines of code you are going to need ahead of time, you can make some educated guesses based on past experience. You should at least take a stab at the worst case, best case, and average case of past scenarios. Then you can take a weighted average of the three (perhaps giving them weights 1, 1, and 3, respectively) to make an "expected scenario."

Feel free to fudge things a bit to take into account any extra information you may have. For example, if a new project is fairly complicated and very different from past projects, you may want to change the weighting factors a bit to give the worst case a bit more pull.

(If you majored in Divinatory Statistics in college, feel free to calculate σ, μ, ρ, and any other Greek letters that you think will help you to better predict the most likely outcome.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Another problem with size-oriented metrics is that they're language-dependent. The same program may require 1,700 lines of code if you write it in assembly but only 750 lines if you write it in Java.

These metrics also penalize programs that use short but elegant solutions. Project Harmony might do the same thing as project Fracas but using half as many lines of code written in the same amount of time. If the result is better designed and more elegant, then it might be better code even though it looks like the Harmony team was one-half as productive.

Function Point Normalization

The real problem with size-oriented normalization is that it's tied to a particular implementation of an application, not to the application's inherent complexity. *Function-point* (*FP*) normalization tries to fix that by calculating a FP number to represent the application's complexity. You then divide various attributes such as lines of code or number of bugs by the FP value to get a normalized result.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

> **ANOTHER BLAST FROM THE PAST** *Function point analysis was developed in the 1970s by Allan J. Albrecht in an attempt to measure application complexity without counting lines of code. Like Ishikawa diagrams, function points are useful enough that they've stuck around.*

Function points measure a project from the user's point of view so they count what the application does, not how it does it. Because they are measured from the user's point of view, they should be hardware-independent and software-independent. An application should do the same things whether you build it in C++ on a Linux desktop system, in Java on an Android device, or in COBOL on a mainframe.

There are many different variations on function points that use various measures of the application's behavior to represent its complexity. For example, different versions count the number of forms, external inputs, event triggers, and so forth. This section describes a version that is reasonably easy to calculate and that seems to be fairly common.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

I'll describe the calculation details shortly. First, here's an overview of the process.

1. Count five specific *function point metrics* that include such things as the number of inputs and the number of outputs.
2. Multiply each of those values by a *complexity factor* to indicate how complicated each activity is. Add up the results to get a *raw FP* value.
3. Calculate a series of *complexity adjustment factors* that take into account the importance of general features of the application. (For example, how important is the transaction rate to the application?) Add the complexity adjustment factors to get the *complexity adjustment value* (*CAV*).
4. Take a weighted average of the raw FP and the CAV and voilà! You get the final FP value.

Don't worry if this seems complicated. It requires a lot of steps, but each of the steps is quite simple. The following sections describe the four main steps in greater detail and show an example calculation.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Count Function Point Metrics

In this step, you estimate the number of the following items.

Inputs—The number of times data moves into the application and updates the application's internal data. This includes inputs the user enters on screens and inputs from other applications and external files. An example would be a New Student form that lets the user enter student ID, name, address, phone, and other information in the application's database.

Outputs—The number of times outputs move out of the application. This includes outputs displayed to the user as well as outputs sent to external systems or external files. An example would be producing a Delinquent Account report that lists accounts with outstanding balances. The report could be printed, sent to an external file, or sent to another program for processing.

Inquiries—The number of times the application performs a query/response action. This is different from an input followed by an output because it doesn't update the application's internal data. For example, the user might enter a customer ID and the application would display that customer's information, but it wouldn't update the database.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Internal Files—The number of internal logical files used by the application. This includes things such as configuration files, data files, and database tables.

External Files—The number of files that the application uses that are maintained by some other program. For example, the application might use an inventory database that is maintained by a separate inventory tracking program.

The next step is to multiply the number of each kind of item by its complexity.

Multiply by Complexity Factors

As you count these metrics, you should estimate the complexity of each. For example, consider as an input a New Student form that lets the user enter information about a new student. Suppose the form contains 15 text boxes. You may decide that means this input has medium complexity.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

COMPLEXITY CONUNDRUM

Different FP techniques use different methods for deciding whether a piece of the system has low, medium, or high complexity. Some look at factors such as the number of internal tables and the number of data values involved in an action. For example, an Order Creation form might create records in three tables and include 20 fields where you enter data.

In this chapter, I'll just assume you can use your intuition to assign complexity values because that's a lot simpler. The exact formula you use doesn't matter too much as long as you're consistent across projects.

If you want to compare the FP values of your applications to those of programs written by other groups, then you need to use one of the more precisely defined methods for determining complexity. (For an example, see the tables at `http://www.softwaremetrics.com/fpafund.htm`.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

			Complexity				
Category	**Number**		**Low**	**Medium**	**High**		**Result**
Inputs	______	×	3	4	6	=	______
Outputs	______	×	4	5	7	=	______
Inquiries	______	×	3	4	6	=	______
Internal Files	______	×	7	10	15	=	______
External Files	______	×	5	7	10	=	______
					Total (raw FP)		______

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

FIGURE 10-4: Use this table to calculate raw FP.

After you calculate a complexity value for each of the items you're counting, use them to get a sense of the overall complexity for each of the metric categories. For example, if you have two low, five medium, and one high complexity inputs, then the inputs as a whole have medium complexity.

Now multiply the number in each metric category by the appropriate values shown in Figure 10-4.

Figure 10-5 shows a sample raw function point calculation. For example, this application has 10 inputs with a relatively high complexity. In the first line of the calculation, the number of inputs 10 is multiplied by the high complexity factor 6 to give the result 60.

			Complexity				
Category	**Number**		**Low**	**Medium**	**High**		**Result**
Inputs	10	×	3	4	(6)	=	60
Outputs	5	×	(4)	5	7	=	20
Inquiries	4	×	3	(4)	6	=	16
Internal Files	23	×	(7)	10	15	=	161
External Files	2	×	(5)	7	10	=	10
					Total (raw FP)		**267**

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

FIGURE 10-5: In this example, the application's raw FP value is 267.

The next step is to apply complexity adjustment factors.

Calculate Complexity Adjustment Value

The function point metrics look at particular facets of the application. The complexity adjustment factors include a series of indicators designed to measure the complexity of the application as a whole.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

C-A-V IS NOT FOR ME

Some developers use the raw FP and don't bother with the CAV. The two main reasons are that some cost estimation tools take the raw FP as an input and that the CAV plays too big a role in the final FP calculation. The web page `http://alvinalexander.com/FunctionPoints/node29.shtml` has more information about this issue (although it uses some different terminology).

To calculate the complexity adjustment factors, consider each of the following items.

1. Data communication
2. Distributed data processing
3. Performance
4. Heavily used configuration
5. Transaction rate
6. Online data entry
7. End user efficiency
8. Online update
9. Complex processing
10. Reusability
11. Installation ease
12. Operational ease
13. Multiple sites
14. Facilitate change

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Rate the importance of each of the 14 factors according to Table 10-3.

TABLE 10-3: CAV Ratings

IMPORTANCE	RATING
Irrelevant	0
Minor	1
Moderate	2
Average	3
Significant	4
Essential	5

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

After you make these decisions, simply add the complexity adjustment factors to get the complexity adjustment value.

Table 10-4 shows a sample complexity adjustment calculation.

TABLE 10-4: Sample CAV Ratings

FACTOR	RATING
Data communication	2
Distributed data processing	4
Performance	5
Heavily used configuration	2
Transaction rate	3
Online data entry	5
End user efficiency	5
Online update	1
Complex processing	1
Reusability	0
Installation ease	4
Operational ease	5
Multiple sites	5
Facilitate change	1
Total (CAV)	**42**

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The final step is to use the raw FP and the CAV to calculate the adjusted FP value.

Calculate Adjusted FP

To calculate the final FP, simply use the following formula.

$$FP = (\text{raw FP}) \times (0.65 + 0.01 \times CAV)$$

For example, the calculation in Figure 10-5 got a raw FP of 267. The CAV in the preceding section was 32. For those values, the final FP result is:

$$FP = (267) \times (0.65 + 0.01 \times 42) = 285.69$$

SUMMARY

Metrics enable you to characterize and track projects. Process metrics let you compare multiple projects over a long period of time to see if you can improve your development process. For example, if one project has fewer bugs per line of code than the others, you can study that project to see why it is different and try to reproduce those results in future projects.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Project metrics enables you to make predictions about a project that is still underway. For example, if a project isn't producing enough lines of code for where it is in its schedule, you can look for ways to increase productivity. Without metrics, it's often hard to tell when a project is going off course until it's too late to do anything about it.

Size normalization enables you to compare projects of different sizes. Function point normalization enables you to compare projects of different sizes and complexities. Comparisons are always better if the projects are similar, but those two techniques enable you to get at least some meaningful information from projects with some differences.

This chapter focused mostly on lines of code and bugs, but the same techniques apply to every output from software engineering. You can use metrics to track the specifications, use cases, design documents, change requests, and number of donuts eaten. If any numbers wander away from what's normal, you can dig deeper to see if there's a problem you can correct or perhaps an unexpected benefit you can exploit in the future.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Tracking bugs is a good way to estimate the application's maintainability. If the code is buggy, then maintaining it will probably be hard. The next chapter focuses on maintenance. It explains what the team's role is during the maintenance phase and describes some of the directions the project can take after its initial release.

EXERCISES

1. Suppose a project has a `States` table that lists the states where the customer does business. A search dialog lets the user select one of the states from a drop-down list to select accounts from the selected state. Some of the use cases call for the states to be set to Maine, Vermont, New Hampshire, and Massachusetts, but during tests New Hampshire doesn't appear in the drop-down list.

   Draw an Ishikawa diagram showing possible causes for this problem. What steps would you take to try to find the root cause of the problem?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

2. Compare size normalization and FP normalization. When would you use one or the other?

3. Are there times when you could use both size normalization and FP normalization to compare two projects?

4. Assume a project has a raw FP score of 500. What are the largest and smallest final FP values the project might have? How would it achieve those values?

5. Give an example where project A has more bugs than project B but seems to be in better shape according to size normalization. Assuming the projects are in roughly the same development stage, what else do you need to know to decide whether project A will finish before project B?

6. For the example you made for Exercise 5, what else do you need to know to estimate when the two projects will finish flushing out all their bugs?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

7. Calculate an FP value for Microsoft WordPad.

8. Calculate an FP value for Microsoft Word.

9. Judging from your experience solving Exercises 7 and 8, how consistent do you think FP values will be when different people perform the calculations? (Compare your solutions to my solutions in Appendix A, "Solutions to Exercises," if you like.) What could you do to improve consistency?

10. What do your solutions to Exercises 7 and 8 tell you about Microsoft WordPad and Microsoft Word? Does that result agree with what you would expect?

11. Which do you think is better, size normalization or function point normalization?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

12. Table 10-5 shows the number of programmers that worked on four projects.

TABLE 10-5: Number of Programmers

PROJECT	# PROGRAMMERS
Unicorn	10
Pegasus	8
Griffin	12
Jackalope	7

Table 10-6 shows the cumulative numbers of lines of code written and bugs discovered during each week of active coding for the four projects. For example, by the end of week 3, project Griffin contained 5,141 lines of code and 62 known bugs.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

TABLE 10-6: Lines of Code and Bugs

	UNICORN		PEGASUS		GRIFFIN		JACKALOPE	
WEEK	LOC	BUGS	LOC	BUGS	LOC	BUGS	LOC	BUGS
1	1,107	0	542	0	450	3	126	5
2	2,349	2	1,374	12	2,392	17	1,201	27
3	3,482	7	2,759	37	5,141	62	3,515	60
4	4,272	30	4,680	61	6,008	102	5,176	72
5	6,009	72	6,012	89	7,817	156		88
6	7,522	110		104	9,750	160		
7	9,759	156				175		
8	11,895	207						
9		273						

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The final bug numbers for each project include bugs found after initial programming stopped.

Assuming these projects have roughly similar complexity, how can you meaningfully compare the programmers' productivity and bug rates at the ends of the projects? What do your calculations show? Can you think of any places to look for process improvements?

13. Suppose you're tracking a new project (project Hydra) that you expect to include approximately 7,000 lines of code. Assuming its progress is similar to the progress of the projects described in Exercise 12, how many person-weeks should this project's programming phase take? How many bugs do you expect to find?

14. Seeing the results of Exercise 13, you decide you can finish the programming for project Hydra in nine weeks with five developers. Table 10-7 shows the project's actual progress after week 3.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

TABLE 10-7: PROJECT HYDRA PROGRESS

WEEK	LOC	BUGS
1	370	0
2	693	2
3	969	12
4	1,251	24

Should you be concerned?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

▸ WHAT YOU LEARNED IN THIS CHAPTER

- You can rate a defect's importance by discoverer, severity, time created, and age at fix.
- Group defects by task (specification, design, programming, hardware, and so forth) to look for common causes.
- An Ishikawa diagram can help you find the root causes of a defect.
- Attributes are things you can measure; metrics are values you can use to evaluate a project; and indicators give indications of a project's state and future.
- Attributes→metrics→indicators→projections and process improvements.
- Software metrics let you characterize, track, and predict a project's characteristics such as defects, bugs, and lines of code written.
- Process metrics are used to improve your development process in the long run. Project (or product) metrics are used to track and predict the current project's progress.
- Size-normalized metrics enable you to compare projects of different sizes but similar complexities. These metrics are values divided by a measure of the project's size. For example, "bugs per KLOC" or "pages of documentation per person."
- Function points enable you to estimate a project's complexity.
- Function point normalization enables you to compare projects of different sizes and complexities. Values are divided by the project's FP value to give metrics such as "KLOC per FP" or "bugs per FP."

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

11

Maintenance

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Troutman's Second Programming Postulate: The most harmful error will not be discovered until a program has been in production for at least six months.

—Anonymous

All programming is maintenance programming, because you are rarely writing original code.

—Dave Thomas

WHAT YOU WILL LEARN IN THIS CHAPTER:

- Why maintenance is a major percentage of total project cost
- The four categories of maintenance tasks

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- What the "second system" and "third time's a charm" effects are
- How to know when you should rewrite code to make it more maintainable
- Bug tracking states

So you finished the initial release of your application and held wrap-up meetings to make your team members better software engineers. Congratulations! On most projects, a fair number of those people will now wander off to do other things. Some will join new projects and start the whole process all over again. Others may take new roles on other projects. For example, a programmer may become a team lead or a team lead may become a project manager. Still others may leave to satisfy their life-long dreams of becoming lobster fishermen.

Hopefully, a few team members will remain as the project moves into the maintenance phase. Having some original team members during maintenance helps provide continuity for the project so that its original vision isn't lost.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

CHANGE FOR THE WORSE

On one project I worked on, the maintenance crew took over and made all sorts of changes to improve the application. Then the users spent weeks forcing them to put things back the way they were. If more of the maintenance group had been around from the beginning, they would have understood the application better and been more hesitant to make those changes in the first place.

Some programmers dislike maintenance programming because they find it boring. It can be a lot of fun to write an application that finds the shortest route that visits all the ice cream stores in your town for a bicycle ice cream crawl. It's less fun to debug your application when you discover it is telling people to bike on the highway. It can be downright painful to dive into someone else's rat's nest of kludges, hacks, and bug fixes to make major changes.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Maintenance may not always be a lot of fun, but it's important because maintenance is often relatively expensive. (In management-speak, maintenance accounts for a large percentage of ***total cost of ownership*** or *TCO*.) Often maintenance accounts for 75 percent of a project's total cost.

This chapter describes the tasks that make up software maintenance. It explains why maintenance is expensive and methods you can use to reduce maintenance costs.

MAINTENANCE COSTS

You may wonder why maintenance is such a large percentage of a project's total cost. One reason is that applications often live far longer than they were originally intended. A typical business application might be in use years or even decades after it was written. Most businesses are stingy, so writing a new program to replace an old one that still works is rarely an option. (Even if the existing application is so old it measures lengths in cubits and lets you click a sundial to select times.) I worked on one application 30 years ago that is still in use today, even though the company that owns the code has been bought at least twice since then.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

For further proof that applications often exceed their life expectancies by decades, consider the Y2K problem. Programs written as far back as the 1950s and 1960s stored dates as 2-digit numbers to save space. For example, the year 1978 was stored as '78. That worked pretty well until after the year 2000 when dates became ambiguous. For example, if you're working for a hospital in 2005 and one of your patients was born in '03, should you schedule a pediatric appointment or a geriatric appointment? I doubt many programmers in 1960 imagined their code would still be in use 40 years later.

Contrary to the predictions of the pundits on the Y2K apocalypse lecture circuit, planes didn't fall from the sky like leaves in autumn; missiles didn't decide they were past their expiration dates and explode; and streets didn't boil with lava. Still, retired BASIC and COBOL programmers from around the world briefly hung up their fishing rods to help the $300 to $600 billion effort to fix this single problem. That's a lot of money to keep decades-old software running. (I'm not sure what will happen when the Y3K problem hits and there are no COBOL programmers left.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The moral is, you should pretend you're carving your code in stone to last for the ages. Chances are it'll last longer than you expect.

A second important reason why maintenance costs often dwarf initial development costs is that it's much easier to write fresh code than it is to modify old code. To safely modify old code, you need to spend time studying it. Because you didn't just write the code, it's not fresh in your mind. If you don't dig into the code and make sure you understand how it works, you're just as likely to add bugs to the code as remove them.

After you make your changes, you need to test them to verify that they work. You also need to thoroughly test the rest of the application to make sure your changes didn't break anything.

You can reduce maintenance costs by doing a good job when you write the initial code. For example, develop simple but flexible designs, use good programming practices, insert comments to make the code easy to read, and provide documentation so future generations of maintenance programmers can figure out what you were thinking when you wrote the code.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

TASK CATEGORIES

At a high level, the tasks that go into long-term maintenance are roughly the same as those that go into initial development. You gather requirements, make designs, write some code, test the code, and deploy a new version. Although the tasks are similar, the focus is different. During maintenance, you tend to spend more time on bug fixes and feature enhancements than on writing completely new code.

Generally maintenance tasks are grouped into the following four categories:

- **Perfective**—Improving existing features and adding new ones
- **Adaptive**—Modifying the application to meet changes in the application's environment
- **Corrective**—Fixing bugs
- **Preventive**—Restructuring the code to make it more maintainable

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The relative effort spent on each of these categories depends on the project. For example, in a relatively small phone app with a short lifespan, you might focus most of your energy on building a new version (perfective) and not worry about making the current version compatible with future phone operating systems (adaptive). For a larger project that you plan to use within your company for many years, you might spend a lot more effort on bug fixes (corrective).

For a typical large application, the relative effort spent on each of the categories (out of the 75 percent of the project's total cost represented by maintenance) might be:

- **Perfective**—50 percent
- **Adaptive**—25 percent
- **Corrective**—20 percent
- **Preventive**—5 percent

The following sections describe these categories in more detail.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Perfective Tasks

For many applications, particularly large ones with long lifespans, this is often the biggest part of maintenance. If you've done a good job building the initial application, the users may like it, but they still want tweaks, adjustments, and improvements. (Although this doesn't include bug fixes, which are tweaks of a different sort. I'll talk about those in the "Corrective" section a bit later in this chapter.)

Sometimes, the specification didn't represent exactly what the users need to do. Maybe the specification didn't explain the user's needs correctly. (Although you should have caught that earlier when the customers reviewed the specifications.) Or maybe the users didn't quite understand what they would need to do after the application was in place.

Sometimes, the tools you built let the users think of ways to do things that they hadn't before. Often the users don't know exactly what's possible until they see the program and have a chance to work with it for a while. They know how they're doing their jobs now, but sometimes no one actually knows how they will do their jobs with your new tools.

Users may also want completely new features that weren't in the original specification. They may have been left out of the first release to save time. Sometimes, it's another example of the "we didn't know this was possible until now" scenario.

Even you and your fellow developers can think of improvements and modifications based on the users' experiences. After you watch the users bashing away on your application, you may discover whole new uncharted areas of new opportunities that the users can't see because they don't have your software engineering background.

THE BIG PICTURE

One of the first big projects I worked on was a dispatching system for telephone repair people. You entered information about jobs and employees, and the system assigned employees to appropriate jobs.

The program included a map that showed all the employees and their jobs on a street map. It didn't help the work assignment code, but we stuck it in anyway, mostly because it had a high "gee whiz" factor for executive presentations (and because it was fun to program).

After the users had been experimenting with the system for a while during parallel testing, we noticed that the dispatchers used a map screen a lot more than we expected. In fact, they used it all the time. We asked them why and they said it was the only place in the system where they could see a list of every employee in the system. We had all sorts of screens that let them look at a particular employee, an employee's assignments, jobs that had not been assigned, and so forth, but no place where they could see every employee's data at the same time.

So we added a screen to do that. (They still used the map screen a lot. I think they just liked it. It also let them see if the employees were all assigned to jobs that happened to be near the same restaurant around lunchtime. That really reduced productivity.)

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The tasks that fall into the perfective category tend to be one of two sorts: feature improvements and new features.

Feature Improvements

Feature improvements involve modifying existing code, so in some ways they're similar to bug fixes. That means you should be aware of the same issues when you handle them.

You need to carefully study the existing code so that you're sure you understand what it does and how it works. You need to plan the modifications you're going to make. Don't just start ripping out old code and typing in new. When you're reasonably certain that your changes won't break things you can make your modifications.

Remember that changing old code is more likely to introduce bugs than writing new code, so you need to test your changes thoroughly. The users probably won't like your modification if it doesn't work or it breaks something else.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

New Features

Adding new features to an application is a lot like writing code for the initial application, so you should follow the same steps:

1. Make a specification explaining what you will do.
2. Get the users to sign off on the specification so that they agree that you're doing the right thing.
3. Create high-level and low-level designs.
4. Write the code.
5. Test, test, and test. (And save the tests in case you need to run them again later.)
6. Use good practices (such as staging and gradual cutover) to deploy the new version of the program.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Adding new features is almost like running a completely new mini-project.

If there are enough changes or the changes are big enough (for example, they require restructuring the program's class hierarchy or architecture), you may want to create a new major version of the application.

A new version is basically a whole new project. Start over from scratch and follow all the steps described up to this part of the book. You can probably take a lot of shortcuts because of your experience with the first version of the program. For example, you may reuse most of the high-level design you wrote for the previous version.

The Second System Effect

In his book *The Mythical Man-Month* (Addison-Wesley, 1975), Frederick Brooks says, "…plan to throw one away; you will anyhow." The notion is that you will learn a lot about the system you need to build when you build it. After you're finished, you'll discover that you could have done a lot of things better, so you throw the first version away and write a new one.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Some developers have suggested that you could crank out a hasty version of the application, throw it away, and then build the real application. Perhaps then you can learn what you need to know without spending all the effort needed to build a "real" first version. (That's sort of what prototyping is.)

Of course if you *plan* to throw away the first version, you may do such a poor job of it that you don't learn the things you need to know to build the good second version. You may use so many sloppy shortcuts that you don't get any practice using the "real" techniques you'll need to build a solid second version.

That leads to the corollary, "If you plan to throw one away, plan to throw two away." If you don't learn anything from the "quick and dirty" first version, your second version is basically just a delayed first version. (By the process of mathematical induction, that means you should plan to throw them all away.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

In the 1995 edition of his book, Brooks retracted his initial assertion, saying it was too simplistic and implicitly assumes you're using the waterfall model of development. (You'll learn more about models of development in Chapters 12 through 14.)

Still, Brooks's notion of a "second system effect" has some merit. The first time you build a system, you don't know everything you'll need to do. You don't necessarily have perfect specifications, and you don't know how to implement the features that are specified correctly. You don't know how the pieces fit together. Sometimes, you may not even know what the pieces are.

When you build the second system, you know a lot more about what you can do and how things need to work. Unfortunately, that sometimes leads developers to throw in every conceivable cool feature (plus the kitchen sink) to make the application the best software solution ever created by programmer-kind. As a result, the second version is confusing, hard to use, bloated, and generally inefficient.

Finally, in the third version (if you have any customers left), you can build the application that you should have built in the first place. At this point, you're a Master Craftsman at software development, programming in general, and your application in particular. You know what the application should do and (just as important) what is should *not* do. You know which user interface features work and which don't. You know what pieces are necessary and how they all fit together. You have become one with the development environment and are perfectly positioned to build the best system possible.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

THIRD TIME'S A CHARM

The third version of an application is often the first version that's really useful. In fact, it's so common that many users wait until the third version before they buy a product. (I've done that several times.)

To prevent users from waiting (and depriving them of much-needed revenue and customer-assisted debugging), some software companies give the first release of a product the version number 3.0, hoping users will buy it. Occasionally, you can find an application version 3.0, but there was never a version 1.0 or 2.0.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

It doesn't always have to work like that. You can struggle against fate and make a real difference, but it takes some effort. To avoid building one or more throwaway versions, you need to carefully follow these steps.

1. Gather requirements, write a specification, and thoroughly validate it with the users.
2. Make high-level designs that provide a framework for development. It should keep pieces loosely coupled and provide enough flexibility to do what you need it to do.
3. Create low-level designs that indicate how to create the features you need.
4. Write code while following good programming practices so that you don't end up with a tangled web of mysterious and uncommented code.
5. Test thoroughly to flush out bugs as quickly as possible.
6. Use good deployment techniques (such as staging and gradual cutover).

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

In other words, follow all the normal steps of software development.

Having developers who are experienced with the type of application you build can also help reduce the "second system" and "third time's a charm" effects. If they've already built their first (and possibly second) versions, you can build more useful versions.

Iterative and RAD development models use other techniques to try to keep development moving toward a useable application. (You'll learn more about them in Chapters 13 and 14.)

Adaptive Tasks

Adaptive tasks help keep the application usable when the things around it change. If the users' hardware, operating system (OS), database, other tools (such as spreadsheets or reporting tools), network security, or other pieces of the users' environment change, it could break your application, so you have to fix it.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Unfortunately, the tools on which your application relies may also be interrelated, so changes to one may affect the others. For example, suppose your application uses a graphics toolkit that uses a particular database. Now the operating system changes so the toolkit no longer works. You upgrade the toolkit so that it's compatible with the new operating system, but the new version of the toolkit isn't compatible with the current version of the database it needs. Unfortunately, the database vendor hasn't finished building a version that's compatible with the new version of the operating system, so you're stuck.

There's no combination of your application, the graphics toolkit, and the database that can run on the new operating system. You can tell the users that it's not your fault, but they still can't process orders, so customers can't get their electric roller skates (or whatever).

To make matters worse, the same scenario can arise if any one of the tools you use is updated. For example, if a new version of the graphics toolkit is released, it may break your application. If a new version of the database appears, it may break the graphics toolkit. Then you're stuck waiting for the graphics vendor to update its toolkit before you can even see if the changes will break your application.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You can take a couple approaches to make these scenarios less likely. First, you can minimize the use of external tools. If you don't use a graphics toolkit, you don't have to worry about a new version breaking your application. If a new version of the operating system breaks your application, at least you have only your own code to fix. You don't need to wait weeks or months for all your tool vendors to revise their products until a workable combination is possible.

Second, you can just ignore new releases of operating systems, databases, toolkits, and any other external tools that you use. I knew one company that had a single computer that ran a program to control a robotic assembly line. Unfortunately, a vendor discontinued support for the programming language used to write the program. After the next operating system release, programs written in that language would no longer be supported. In some later operating system version, the program would stop working completely.

As if that weren't bad enough, new computer hardware wouldn't support the older version of the operating system needed by the program. (Just try to buy a modern computer running Windows 3.11 or OS/2.) Eventually, the computer running the program would die, and the assembly line would be offline for good.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The company could rewrite the application in a new language, but that would be a lot work (in other words, expensive). Besides, the program did what the company wanted and didn't need any new features.

To solve the problem cheaply, the company bought some inexpensive computers, installed the current operating system on them, and added the assembly line program. Now when the computer controlling the assembly line dies, the company pulls another computer from the closet and the assembly line is back up and running. (Although the company is sort of stuck in the 1009s. Eventually that program is going to need to be rewritten.)

Ignoring upgrades worked for that company, but it's a strategy that's getting harder to follow. These days many products install new releases automatically, so it's harder to avoid having some product upgrade itself and break something.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Some companies fight that problem by explicitly prohibiting any upgrades. When a new version of the operating system or some other important piece of the environment is available, the IT department loads the latest version of everything on an isolated computer and tests it. If everything works, the new configuration is rolled out to the users' computers. (Remember staging from Chapter 9, "Deployment"?)

This tight control over the users' computers often seems arbitrary and totalitarian to the users. (Why can't I install the latest version of Othello on my computer?) But you can understand what they're trying to accomplish.

Corrective Tasks

Corrective tasks are simply bug fixes. You've probably been making them since you started development, if not sooner. If you think of mistakes in the specification, designs, documentation, and other pieces of the program as bugs (and you should), then you've been fixing bugs since the project started. (If you extend that a bit to the world outside of work, then you've been fixing bugs since the day you were born. For example, being unable to walk and talk is a bug that takes a newborn about two years to fix.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You probably already know in general how to fix a bug. Find it, study the code so that you're sure you understand it, fix the bug, test, and release a new version of the application with the bug fixed.

There are several ways this process can go wrong. Perhaps the most obvious way is if you fix the bug incorrectly and don't notice during your tests. In that case, you can add an additional step at the end: file another bug report describing the mistake you made fixing the original bug.

One of the worst ways to fail to fix a bug is to lose track of it. At least if you fix a bug incorrectly you have some record of the original bug. If you lose track of a bug, there's little chance that it will ever be fixed (unless a user reports it again). What may be just one of dozens or hundreds of bugs to you, might be really important to some user patiently waiting for you to fix it.

To avoid losing bugs, you need a bug tracking system. There are several kinds of bug tracking systems you can use.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

INCIDENTALLY

Some companies don't like the term "bug" (maybe they're insectophobic), so they call these things "incidents." They may also include change and enhancement requests in the incident category, possibly so that they can deemphasize the number of bugs. (Sure we have 3,000 incidents, but lots of them are change requests.)

For really small projects, you can simply keep track of bugs in a spreadsheet. That works well if you don't have too many bugs but doesn't work as well if you have many developers who need to update the spreadsheet at the same time.

You can store bug information in a directory hierarchy. You would create a text file describing each bug and then move them from directory to directory to group them by status. For example, the `Bugs/New` directory (or `Bugs\New` if you have a Windows accent) would contain new bugs that have not yet been examined by the project team. The `Bugs/Assigned/Rod` directory would contain bugs assigned to me.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

You can store bug information in a database. That works well, but building the database and tools to work with it can be a lot of work. If you're not a database developer who thinks of this as a fun exercise to crank out over the weekend while your friends are off waterskiing, you might want to use a prebuilt bug tracker instead.

Finally, you can use a prebuilt bug tracking application. There are a lot of bug tracking applications available ranging in price from $0 to a $1,000 or so per month. (The expensive ones are designed for *really* big projects with up to a few thousand users.)

Whichever method you use, you can assign a ***state*** to each bug to keep track of its status within the system. The following list describes typical states that a bug tracking system might use:

- **New**—The bug has just arrived and has not yet been assigned to anyone.
- **Assigned**—The bug has been assigned to someone to fix.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- **Reproduced**—The bug has been reproduced by a team member. The bug's description includes instructions for reproducing the bug. (It is sometimes called "verified.")
- **Cannot Reproduce**—A team member has examined the program and can't make the bug occur. Often a "we can't reproduce the bug" message is sent back to the customer and the bug is closed. (As a user, this is my least favorite status because I know the vendor will never fix this bug. Often it seems like they didn't try very hard to reproduce it.)
- **Pending**—A request for more information has been sent to the customer who reported the bug. Sometimes, this state is used before Cannot Reproduce.
- **Fixed**—The bug has been fixed but not tested yet. (This is sometimes called "resolved.")
- **Tested**—The fix has been thoroughly tested and the bug is verified as gone.
- **Deferred**—The bug should not be fixed, or at least not yet. For example, you might want to fix it in the next major release because fixing it now would be too hard.
- **Closed**—The bug has been either fixed, deferred, or otherwise abandoned (see the Cannot Reproduce status), and no further action will be taken on it. (This is sometimes called "resolved.")
- **Reopened**—The bug reappeared after being closed. The bug should probably be treated as if it were a new bug. Although there may be some benefit to reassigning it to the person who originally worked on it, because that person may know more about the code containing the bug. (And it seems like a fitting punishment.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Bug tracking applications typically come with an assortment of features. For example, they may produce reports showing bugs in various states, bugs cleared over a period of time, and bugs assigned to a particular developer. Some can notify developers via e-mail or some other method when bugs are assigned to them.

Some systems can automatically move bugs from one state to another. For example, when a new bug report appears, the system might assign it to a developer chosen from a list of those who are allowed to fix bugs. It would assign the bug to that person and change the bug's status to Assigned.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Another approach would be to have the system send the bug to a manager and ask the manager to assign the bug to a developer.

Some systems also allow you to indicate who is allowed to make certain transitions. For example, if you have separate bug fixers and testers, you could allow the fixers to move a bug from Assigned to Fixed, and the testers would move the bug from Fixed to Tested (or send it back to Reproduced if the tests fail).

MY FAVORITE BUG

My favorite bug of all time appeared in a large application we developed for internal company use. One of its forms let the user enter search criteria to build a list of matching jobs. The user could then scroll through the list and double-click on jobs to get more information.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

> During testing, one of our power users reported that this form occasionally made the application crash. She didn't know why and could not reproduce the crash reliably, but it happened about once per day.
>
> We performed dozens of test queries, opened multiple detail screens, resized the form, and did everything we could think of to test the program, but we couldn't make the crash happen even once.
>
> Finally, we both flew to Tampa (she from Fort Wayne, I from Boston) so I could watch her crash the program. For about half an hour, she built lists, clicked buttons, rearranged panels, and resized the form. Just as she was starting to doubt her own sanity, the program crashed. It took me about another half an hour to figure out how to reproduce the crash.
>
> It turned out that the form contained a splitter that you could use to make one panel smaller (holding search criteria) and another larger (holding query results). There was a bug in the splitter control we were using that made it crash if you made one of the panels exactly 1 pixel tall. It was pretty hard to do. If the panel was 2 pixels tall, everything was fine. You had to be resizing the panels more or less randomly and release the mouse at just the right instant.
>
> When I could reproduce the problem, it didn't take long to figure out what was happening and how to fix it. I added a quick test in the code to ensure that the panels were never less than 2 pixels tall and everything worked fine.
>
> The point of this story (aside from showing off my ninja debugging skills) is that users are rarely crazy. If you can't reproduce a bug, that doesn't mean it isn't there. It is almost surely there, you just can't find it.
>
> So if you can't instantly reproduce a bug, dig a bit deeper. By all means ask for more information, but don't be surprised if the user doesn't have any more information for you. Then dig even deeper. If you close the bug now, it'll come back to haunt you like the last Easter egg that you didn't find until June—and with the associated smell.

One final twist to an already complicated situation is priority. Some bugs are more important than others. If one bug makes the application crash every day or two and a second bug is a typographical error in the Swedish version of the program, the first bug probably deserves higher priority. (Although a typo may be easy to fix, so you might want to bang it out to boost your productivity stats.)

Preventive Tasks

Preventive tasks involve restructuring the code to make it easier to debug and maintain in the future. The fancy "impress them at cocktail parties" word for rewriting code to improve it is *refactoring*.

If you've been paying the least bit of attention, you know that modifying code is more likely to introduce new bugs than writing new code is. If that's true, then why would you ever mess around inside working code? That would be like asking a mechanic with questionable skills to rebuild your

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

brand new car's engine. It will be expensive, might not make things any better, and might make them a whole lot worse.

If you look back at the "Task Categories" section earlier in this chapter, you'll see that typically only approximately 5 percent of a project's maintenance cost is spent on preventive tasks. That number is low largely because of the risk involved with modifying working code. (Companies also usually have a strong "if it ain't broke, don't fix it" bias, which is completely justified here.)

WHAT'S PREVENTIVE?

The discussion of preventive maintenance doesn't include adaptive tasks (modifying or adding features) or corrective tasks (fixing bugs). Those often require some revision of the existing code. For example, you might need to rewrite a piece of code to add a new feature to the program. If you make the fewest changes possible, that doesn't count as preventive maintenance.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

I'm also not counting rewriting a piece of code right after you wrote it to make it more elegant and flexible. That's part of the initial programming and, yes, you should do it. Code is often pretty rough the first time around, and you can often make it more efficient, easier to modify, and easier to understand if you rewrite it right away. After you write a piece of new code, look it over (by yourself or in a code review) and see if you can improve it before moving on.

This section focuses on changes you make to the code before you need to modify it to make it easier to deal with later.

Despite the dangers, there are several reasons why you might want to refactor code, and a few reasons not to. The following sections describe some of the most important of those reasons.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Clarification

If a piece of code is confusing, you should add comments to it explaining how it works. You should do that as you're writing the code or immediately after you finish writing it, while the code is still fresh in your mind. Later, when people need to read the code (including you after you've forgotten how it works), they have a chance of understanding how the code works.

Unfortunately, many programmers don't include enough (or any) comments. Sometimes, they think the code is obvious because it's still fresh in their minds. Sometimes, they get pulled away for more urgent tasks before they get around to writing comments and documentation. Sometimes, they're just plain lazy. In those cases, it may be worth adding comments to particularly confusing pieces of code.

It's often not worth the effort to add comments to random pieces of code. To write good comments, you need to spend a lot of time studying the code carefully so that you're sure you understand how it works. If you don't know what the code is doing, you may insert misleading comments and they can do more harm than good.

For that reason, I recommend that you add comments to code only when you need to modify it. That gives you extra incentive to study the code carefully (or your modifications won't work). You'll also need to test your changes to verify that they work, and that helps verify your understanding.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

No matter how thoroughly you study the code, however, there's still a chance that you don't really get it, so your new comments should always be regarded with a bit of suspicion. Keep the following quote in mind.

> *Don't get suckered in by the comments—they can be terribly misleading. Debug only code.*
>
> —Dave Storer

That doesn't mean comments are completely useless, but they aren't always correct. That's particularly true for comments added long after the code was written.

Code Reuse

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Sometimes when you're modifying code you realize you've done something similar before. Instead of repeating yourself, it may make more sense to extract the common code into a new class or method that you can call from multiple locations. Then when you need to do the same thing a third, fourth, or fifth time, you won't need to write the same code all over again.

Saving you the trouble of writing repeated code is nice, but the real benefit here is in maintaining the duplicated code. Suppose you have the same (or similar) piece of code repeated in several places. What happens if you find a bug in that code? Or if you just decide to change the way the code works? Perhaps you decide to store values in meters instead of feet or you store some data in a database instead of a file.

In those cases, you need to modify the code in exactly the same way in every place that it occurs throughout the program. If you miss any of the occurrences or make one of them incorrectly, the code will be inconsistent. The resulting bugs can be extremely hard to find. (I know that from firsthand experience.)

Making changes consistently is even harder if the pieces of code are similar but not exactly the same because it makes it harder to find the related pieces of code.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The *DRY* (don't repeat yourself) ***principle*** says you should extract common code any time you repeat yourself.

THERE'S NO SUCH THINGS AS 2

One of my mottos is, "There's no such thing as 2." You can write a piece of code once. If you write it a second time, how do you know you won't then need to write it a third or a fourth time?

The same idea holds for user interface design. A customer record can hold a single emergency contact phone number, but if users decide they want to allow a second, how do you know they won't then ask for a third, fourth, and fifth?

Start by assuming users can have only one emergency contact number. As soon as users want to add a second, assume the customer might have any number of phone numbers. That way you can modify the user interface once instead of several times.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Improved Flexibility

Sometimes, when you modify a piece of code, you realize the code isn't as flexible as you'd like. It was written in a way that made sense at the time, but that prevents you from easily making the changes you now need to make. If you need to make only a single change, you can simply make the change, test it, fix anything you've broken, and move on to the next item on your to-do list.

However, suppose you're going to make similar changes in the future. In that case, it may be worth spending a little extra time now to clean up the code so that it's easier to make those changes later. (This is a new version of "Fool me once, shame on you. Fool me twice, shame on me.")

For a concrete example, suppose you've written a perfectly good application that lets the user control the lights in a high-rise office building. It lets you turn individual lights on and off so that you can use the building as a 35-story pixel display. After the first release, the customers decide they want a tool that lets them display the company logo. Unfortunately, you didn't plan for that, so adding it is a bit of a hassle.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

If you were paying attention when the preceding section talked about the DRY principle and "there's no such thing as 2," you can probably guess where this story is headed. After you write this tool, the customers are probably going to want another one. They'll want new tools to draw letters, words, and simple pictures. (They probably won't ask for scrolling messages and animation just yet because the lights don't turn on and off fast enough to make those work very well.)

You could just write the logo tool, but it would probably be worthwhile to spend a little extra time to refactor the code a bit to make building these sorts of tools easier. It'll cost you more time now, but will save time down the road. More important, it will make the code cleaner, so you'll be less likely to introduce bugs when you write new tools later.

Bug Swarms

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

As mentioned in Chapter 8, "Testing," bugs tend to travel in swarms. What that means is some methods, modules, or classes tend to be buggier than others. That can happen for several reasons. Perhaps that chunk of code is exceptionally complicated or confusing. Perhaps it wasn't thought out well in advance during high-level and low-level design. Perhaps the code was modified several times so its original elegant design has been shredded into confetti. Perhaps the code was written by a beginning programmer who hadn't learned about `for` loops yet.

However they form, bug swarms are dangerous. A piece of code has produced a lot of bugs in the past and is likely to continue spawning bugs in the future. At some point, it's better to step back, study the code so that you understand what it's supposed to do, and rewrite it from scratch.

This can be risky. Buggy as it is, the code probably does more or less what it's supposed to do, so there's a chance you'll replace ugly but working code with elegant but broken code.

That means you should perform at least a quick cost-benefit analysis to decide whether it's worth the risk. For example, if you've wasted two or three hours per week for the last few months fixing bugs in a 30-line method, it's probably worth rewriting that method. In contrast, it's less clear whether you should completely rewrite a `Customer` class that contains 7,000+ lines of code just to chase a couple bugs that you haven't been able to reproduce.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

My rule of thumb is, if I'm sick and tired of fixing bugs in a particular piece of code, then it's time to consider rewriting it.

Bad Programming Practices

Fixing bad programming practices is both a good reason and a bad reason to refactor code. It's a good reason because the result can be code that is easier to understand, test, debug, and modify. It's a bad reason because, in theory at least, you shouldn't have any bad programming practices in your code.

Ideally after you write a piece of code, you should review it (either yourself or in a formal code review) and make sure you've followed good programming practices. Still, sometimes bad code slips into a project.

Even if the code starts out good, it can be modified and remodified until it no longer follows good programming practices. Over time a method that was initially short, elegant, and tightly-focused can morph into an incomprehensible jungle of loops, branches, and unrelated tests.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

For example, suppose you write a nice, short, tightly focused method that takes a student ID as a parameter and fetches that student's data from a database. A few days later, the method's requirements change to make it select only students that are currently enrolled in classes. You modify the code to add that check. No big deal. A week after that, someone else wants to search for students by name instead of student ID, so you add a student name parameter and modify the code accordingly. After a few other changes, the method that once was straightforward is a confusing mess. You used to be able to view all of the code on a single screen, but now it contains hundreds of lines of code with `if-then` statements spanning several pages, so you have to scroll back and forth to figure out what's happening.

At that point, you should probably rewrite the method to restore its original tight focus. You should break this frankenmethod into several separate methods, each of which performs a single, unambiguous task. The result will be more methods and probably a larger total number of lines of code, but the methods will each be easier to understand, use correctly, debug, and maintain.

The following list shows some of the bad programming practices that you should avoid initially and that might indicate a class, method, or other piece of code could benefit from refactoring:

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- The code is too long.
- The code is duplicated.
- A loop is too long.
- Loops are nested too deeply.
- It doesn't do much.
- It's never used.
- It has a vague or unfocused purpose.
- It performs more than one task.
- It takes a lot of parameters.

If you see code that has changed over time to include some of these symptoms of bad code, consider refactoring.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Individual Bugs

Finding an individual bug is not a good reason to rewrite code. If you find a single bug in a method, just fix it and move on. One bug does not make a swarm. There's no reason to rip a good piece of code apart and risk breaking something just because it contains a single bug.

Now if you discover another bug in the same piece of code next week and a third bug the week after that (and you didn't add them while fixing the first bug), then you should think about rewriting the code to clean it up.

Not Invented Here

This is the worst reason for rewriting code, but it's also probably the most common. When some programmers see someone else's code, it doesn't look quite right. The structure is wrong, the names of the variables are misleading (it should use `num_students` instead of `student_count`), the comments don't use proper grammar, and the indentation is off. Terrible code! Who wrote this gibberish?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

The problem isn't actually the code; it's that the second programmer didn't write it. Everyone thinks their approach, naming convention, commenting style, and everything else is the best. If you weren't using the best possible techniques for writing code, you'd do things differently. Thinking you need to rewrite a piece of code just because someone else wrote it is called the ***not invented here syndrome*** (*NIHS*).

When you're in school, assignments tend to have a single correct answer. In contrast, in the real programming world there's ***never*** a single correct answer. There are always several (perhaps hundreds) of different ways to accomplish the same thing. Some ways may be better than others (searching a database by using an index is a lot faster than pulling up random records hoping to get the one you want), but there are usually lots of ways that are good enough to get the job done.

When you see a piece of strange code, you shouldn't ask yourself whether it's the correct solution or the best solution. Instead you should ask whether it satisfies the criteria that mean it should be rewritten. Just because it's not the solution you would have chosen doesn't mean it needs to be changed.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

That leads to one of my favorite mottos (which I learned the hard way).

> *If it's good enough, it's good enough.*

If the code works correctly, is fast enough to satisfy your needs, and doesn't contain a swarm of bugs, leave it alone.

TASK EXECUTION

Whether you need to modify existing code for perfective, adaptive, corrective, or preventive reasons, you need to follow roughly the same steps to make useful changes without adding new bugs. At a high level, the steps you follow are roughly the same as those that go into initial development:

- Requirement gathering
- High-level design

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

- Low-level design
- Development
- Testing
- Deployment

For smaller maintenance tasks, you can probably abbreviate or skip some of those steps. For example, if you need to change only a single line of code to fix a bug, your requirement gathering probably just includes the statement, "Fix the bug." You also probably don't need to spend a lot of time on high-level or low-level design, and you may defer deployment until the application's next major release.

You should never skimp on testing, though.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

After you make your changes, you need to perform maintenance on your maintenance. (New features you add may contain bugs or need future modification. Any bugs you fix may be fixed incorrectly and need further repair. Of course, those bug fixes may need fixes, which need more fixes, and so on until you wonder if you're trapped in the movie *Inception*. You better keep your chess piece or spinning top handy.)

Some maintenance tasks may be similar to regular development tasks, but their relative frequency often changes. In an application for internal use that does a good enough job already, the focus may be on bug fixing instead of feature improvements and enhancements. That's also often the case with first releases of consumer applications. No one is going to buy version 2.0 if the customers universally hate version 1.0 for its bugginess.

If you sell your application online and need a continuing stream of revenue to keep the creditors at bay, you may decide to focus more on creating new and improved features for a new release. (Of course, you still need to fix any bugs. See the last sentence in the preceding paragraph.)

SUMMARY

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

Maintenance is somewhat similar to normal development. You still need to perform roughly the same tasks (requirement gathering, high-level design, low-level design, write code, test, and deploy the results). Sometimes the focus is slightly different (you'll probably spend more time fixing bugs than writing new code) and you might skimp on some of the steps (you probably won't need an extensive high-level design to fix a one-line bug), but the basic approach is similar. Testing is particularly important so that you don't introduce too many new bugs when you fix old ones.

This chapter finishes the introduction to basic software engineering tasks. All software development projects include the basic tasks in one form or another with varying amounts of emphasis.

The chapters in the next part of the book describe different models of software development. For now, you can think of them as different ways to arrange the basic development steps. For example, iterative models use the same steps but repeated many times to try to keep the project moving toward a usable result.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

EXERCISES

1. Suppose your programming team writes an application with 10,000 lines of code. During testing, you decide that the team generates roughly 20 bugs per KLOC (kilo-lines of code) for new code. (That's probably a bit on the low side for a typical development team, but I'll give you the benefit of the doubt.) During bug fixing, you discover they generate about twice that many bugs when they modify older code. How many lines of code will the team actually generate including original code, fixes, fixes to fixes, and so forth?

2. After you write the lines of code you predicted in your answer to Exercise 1, are you done with maintenance?

3. Consider your answer to Exercise 1. Suppose the number of lines of code the team members can write for different kinds of code is given in the following table.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

CODE TYPE	LINES PER DAY
New code	20
Fix a bug	4
Fix a bug fix	2
Fix a fixed bug fix	1

How many person-days will it take to write all the code?

4. If you have two team members, approximately how many months will the project described in Exercise 3 take? (Yes, I'm totally cheating here. You can look back on a project and calculate the number of lines of code per day you wrote, but you generally can't use imaginary productivity numbers to predict a project's duration.) What if you have five team members? 10? 111?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

5. Draw a flowchart showing how a bug report might move through the states New, Assigned, Reproduced, Cannot Reproduce, Fixed, Tested, and Closed. Allow the bug to move into Pending if it cannot be reproduced.

   Label the connecting arrows with the tasks that lead to the new state. For example, label the arrow leading from New to Assigned "Assign." Require approval before moving a bug into the Closed state.

6. From which states could a bug move into the states Pending, Deferred, or Reopened?

7. Why might you want to move a bug from the Closed state to the Deferred state?

8. What are the total (approximate) percentages of cost spent on each of the four maintenance categories over the life of an application?

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9. Place the following situations in their correct maintenance task categories (perfective, adaptive, corrective, or preventive).

    a. Change the `SaveSnapshot` method because it isn't saving files in BMP format correctly.

    b. Change the `SaveSnapshot` method so that it can also save files in PNG format.

    c. Change the main program to restore the settings in use when the previous session ended.

    d. Add comments to a 200-line method that currently has the single comment `CodeNinja was here 4/1/2003`.

    e. Write documentation to clarify a module's low-level design.

    f. Remove the `PremiumCustomer` class because it isn't used.

    g. Add icons to display on the new operating system's startup page.

    h. Rewrite a method because the program grew so large the old version wasn't fast enough.

    i. Rewrite a method because it uses an unnecessarily complicated algorithm.

    j. Rewrite a 15-line method because it contained seven known bugs (now fixed).

    k. Rewrite a 715-line method to make it smaller.

    l. Rewrite a complicated method to see how it works. Then tuck the new code away for future reference but don't replace the original code in the application.

    m. Rewrite the logging method to use cloud services to store data instead of storing data locally.

    n. Rewrite the login screen to deal with the company's new firewall.

    o. Change the Order List screen to let the user sort orders on any field.

10. In which of the following situations should you consider rewriting a piece of code (preventive maintenance)? If you can't tell from just the description, what else would you need to know before deciding?

    a. A method is 412 lines long.

    b. A method is 10 lines long but very confusing.

    c. A method uses `for`, `while`, and `for-each` loops nested 12 levels deep.

    d. At one point the code makes a series of method calls 37 levels deep.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

e. A method violates the team's variable naming conventions.

f. A method draws a rectangle, line, or ellipse depending on its parameters.

g. It takes 43 seconds to log in to the application.

h. You've just discovered the fifth bug in a 40-line method.

i. Roughly once per day, the program crashes and loses any work the user hasn't already saved.

j. When the user tries to close the application, it crashes. Otherwise it seems to work just fine.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

k. A coworker (definitely not you) fixed a bug in a method, but you later discovered that the bug fix caused another bug. The coworker fixed it again, but that also caused another bug.

l. A coworker fixed a bug in a method and that caused another bug. A different coworker fixed that bug, but the fix lead to yet another bug. A third coworker fixes the latest bug and (you guessed it) caused another bug.

m. Roughly once per day, the program crashes, but the user can easily restart it without losing any work.

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

▶ WHAT YOU LEARNED IN THIS CHAPTER

- Maintenance is *expensive*, sometimes accounting for up to 75 percent of a project's total cost.
- One reason why maintenance is expensive is that applications often live far longer than expected.
- Maintenance tasks can be divided into four categories:
    - Perfective tasks improve, modify, or add features to a project.
    - Adaptive tasks modify an application to work with changing conditions in the environment such as a new operating system version or changes to external interfaces.
    - Corrective tasks are bug fixes.
    - Preventive tasks (refactoring) modify the code to make it easier to maintain in the future.
- Sometimes developers learn what they need to do to build a system while making the first version, so the second version is the first good one (the second system effect).
- Sometimes a system's second version is bloated and full of unnecessary bells and whistles, so the third version is the first good one (the third time's a charm effect).
- Bugs typically travel through some of the following states: New, Assigned, Reproduced, Cannot Reproduce, Pending, Fixed, Tested, Deferred, Closed, and Reopened.
- You don't always need to refactor code, even if it doesn't follow good programming guidelines. (If it ain't broke, don't fix it.)
- To perform maintenance tasks successfully, you need to follow the normal software engineering steps: requirement gathering, high-level design, low-level design, development, testing, and deployment. (Although you can often abbreviate some of those steps. You probably don't need extensive high-level design to fix a one-line bug.)

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

9b3c108192e5fcb5ac110e4bbde32ac1
ebrary

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