9 Software Maintenance: Enhancing Legacy Software Using Refactoring and Agile Methods

Butler Lampson (1943–) was the intellectual leader of the legendary Xerox Palo Alto Research Center (Xerox PARC), which during its heyday in the 1970s invented graphical user interfaces, object-oriented programming, laser printing, and Ethernet. Three PARC researchers eventually won Turing Awards for their work there. Lampson received the 1994 Turing Award for contributions to the development and implementation of distributed personal computing environments: workstations, networks, operating systems, programming systems, displays, security, and document publishing.

There probably isn't a "best" way to build the system, or even any major part of it; much more important is to avoid choosing a terrible way, and to have clear division of responsibilities among the parts.

—Butler Lampson, *Hints for Computer System Design,* 1983

9.1 What Makes Code "Legacy" and How Can Agile Help?

1. Continuing Change: [software] systems must be continually adapted or they become progressively less satisfactory

—Lehman's first law of software evolution

As Chapter 1 explained, ***legacy code*** stays in use because it *still meets a customer need,* even though its design or implementation may be outdated or poorly understood. In this chapter we will show not only how to explore and come to understand a legacy codebase, but also how to apply Agile techniques to enhance and modify legacy code. Figure 9.1 highlights this topic in the context of the overall Agile lifecycle.

Maintainability is the ease with which a product can be improved. In software engineering, maintenance consists of four categories (Lientz et al. 1978):

- Corrective maintenance: repairing defects and bugs
- Perfective maintenance: expanding the software's functionality to meet new customer requirements
- Adaptive maintenance: coping with a changing operational environment even if no new functionality is added; for example, adapting to changes in the production hosting environment
- Preventive maintenance: improving the software's structure to increase future maintainability.

Practicing these kinds of maintenance on legacy code is a skill learned by doing: we will provide a variety of techniques you can use, but there is no substitute for mileage. That said, a key component of all these maintenance activities is ***refactoring***, a process that changes

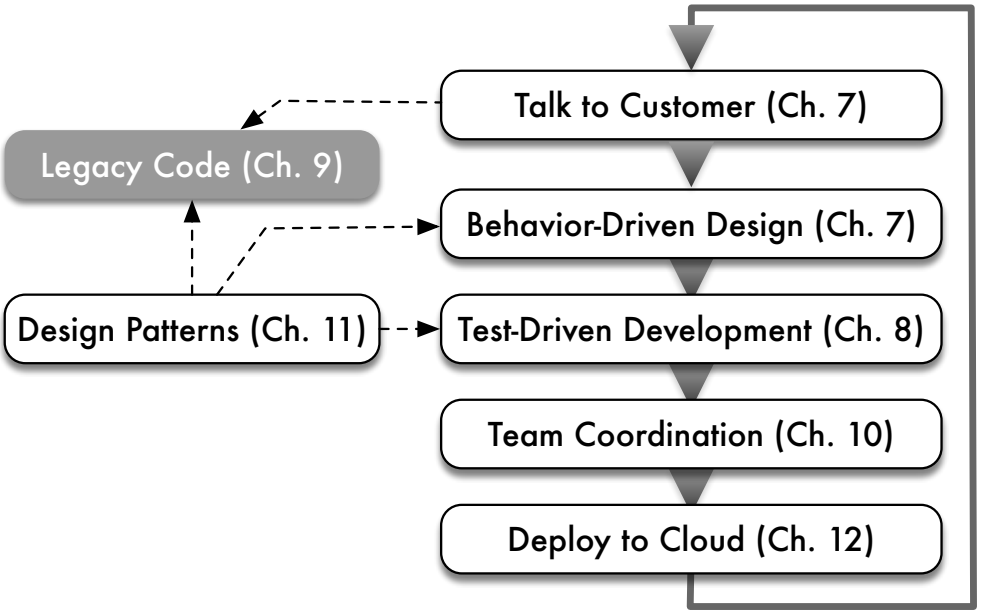

Figure 9.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter covers how Agile techniques can be helpful when enhancing legacy apps.

Highly-readable unit, functional and integration tests (Chapter 8)	Git commit log messages (Chapter 10)
Lo-fi UI mockups and Cucumber-style user stories (Chapter 7)	Comments and RDoc-style documentation embedded in the code (Section 9.4)
Photos of whiteboard sketches about the application architecture, class relationships, etc. (Section 9.2)	Archived email, wiki/blog, notes, or video recordings of code and design reviews, for example in Campfire[1] or Basecamp[2] (Chapter 10)

Figure 9.2: While up-to-date formal design documents are valuable, Agile suggests we should place relatively more value on documentation that is "closer to" the working code.

the structure of code (hopefully improving it) without changing the code's functionality. The message of this chapter is that *continuous refactoring improves maintainability.* Therefore, a large part of this chapter will focus on refactoring.

Any piece of software, however well-designed, can eventually evolve beyond what its original design can accommodate. This process leads to maintainability challenges, one of which is the challenge of working with legacy code. Some developers use the term "legacy" when the resulting code is poorly understood because the original designers are long gone and the software has accumulated many ***patches*** not explained by any current design documents. A more jaded view, shared by some experienced practitioners (Glass 2002), is that such documents wouldn't be very useful anyway. Once development starts, necessary design changes cause the system to drift away from the original design documents, which don't get updated. In such cases developers must rely on *informal* design documents such as those that Figure 9.2 lists.

How can we enhance legacy software without good documentation? As Michael Feathers writes in *Working Effectively With Legacy Code* (Feathers 2004), there are two ways to make

changes to existing software: *Edit and Pray* or *Cover and Modify*. The first method is sadly all too common: familiarize yourself with some small part of the software where you have to make your changes, edit the code, poke around manually to see if you broke anything (though it's hard to be certain), then deploy and pray for the best.

In contrast, *Cover and Modify* calls for creating tests (if they don't already exist) that cover the code you're going to modify and using them as a "safety net" to detect unintended behavioral changes caused by your modifications, just as regression tests detect failures in code that used to work. The cover and modify point of view leads to Feathers's more precise definition of "legacy code", which we will use: *code that lacks sufficient tests to modify with confidence, regardless of who wrote it and when.* In other words, code that you wrote three months ago on a different project and must now revisit and modify might as well be legacy code.

Happily, the Agile techniques we've already learned for developing new software can also help with legacy code . Indeed, the task of understanding and evolving legacy software can be seen as an example of "embracing change" over longer timescales. If we inherit well-structured software with thorough tests, we can use BDD and TDD to drive addition of functionality in small but confident steps. If we inherit poorly-structured or undertested code, we need to "bootstrap" ourselves into the desired situation in four steps:

1. Identify the *change points*, or places where you will need to make changes in the legacy system. Section 9.2 describes some exploration techniques that can help, and introduces one type of Unified Modeling Language (UML) diagram for representing the relationships among the main classes in an application.

2. If necessary, add ***characterization tests*** that capture how the code works now, to establish a baseline "ground truth" before making any changes. Section 9.3 explains what these tests are and how to create them using tools you're already familiar with.

3. Determine whether the change points require ***refactoring*** to make the existing code more testable or accommodate the required changes, for example, by breaking dependencies that make the code hard to test. Section 9.6 introduces a few of the most widely-used techniques from the many catalogs of refactorings that have evolved as part of the Agile movement.

4. Once the code around the change points is well factored and well covered by tests, make the required changes, using your newly-created tests as regressions and adding tests for your new code as in Chapters 7 and 8.

Summary of how Agile can help legacy code:

- Maintainability is the ease with which software can be enhanced, adapted to a changing operating environment, repaired, or improved to facilitate future maintenance. A key part of software maintenance is refactoring, a central part of the Agile process that improves the structure of software to make it more maintainable. Continuous refactoring therefore improves software maintainability.
- Working with legacy code begins with exploration to understand the code base, and in particular to understand the code at the *change points* where we expect to make changes.
- Without good test coverage, we lack confidence that refactoring or enhancing the code will preserve its existing behavior. Therefore, we adopt Feathers's definition—"Legacy code is code without tests"—and create characterization tests where necessary to beef up test coverage before refactoring or enhancing legacy code.

■ ***Elaboration: Embedded documentation***

RDoc is a documentation system that looks for specially formatted comments in Ruby code and generates programmer documentation from them. It is similar to and inspired by JavaDoc. RDoc syntax is easily learned by example and from the Ruby Programming wikibook[3]. The default HTML output from RDoc can be seen, for example, in the Rails documentation[4]. Consider adding RDoc documentation as you explore and understand legacy code; running `rdoc .` (that's a dot) in the root directory of a Rails app generates RDoc documentation from every `.rb` file in the current directory, `rdoc -help` shows other options, and `rake -T doc` in a Rails app directory lists other documentation-related Rake tasks.

Self-Check 9.1.1. *Why do many software engineers believe that when modifying legacy code, good test coverage is more important than detailed design documents or well-structured code?*

◇ Without tests, you cannot be confident that your changes to the legacy code preserve its existing behaviors. ■

9.2 Exploring a Legacy Codebase

> *If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.*
>
> —Rob Pike

The goal of exploration is to understand the app from both the customers' and the developers' point of view. The specific techniques you use may depend on your immediate aims:

- You're brand new to the project and need to understand the app's overall architecture, documenting as you go so others don't have to repeat your discovery process.

- You need to understand just the moving parts that would be affected by a specific change you've been asked to make.
- You're looking for areas that need beautification because you're in the process of porting or otherwise updating a legacy codebase.

Just as we explored SaaS architecture in Chapter 3 using height as an analogy, we can follow some "outside-in" steps to understand the structure of a legacy app at various levels:

1. Check out a scratch branch to run the app in a development environment
2. Learn and replicate the user stories, working with other stakeholders if necessary
3. Examine the database schema and the relationships among the most important classes
4. Skim all the code to quantify code quality and test coverage

Since operating on the live app could endanger customer data or the user experience, the first step is to get the application running in a development or staging environment in which perturbing its operation causes no inconvenience to users. Create a *scratch branch* of the repo that you never intend to check back in and can therefore be used for experimentation. Create a development database if there isn't an existing one used for development. An easy way to do this is to clone the production database if it isn't too large, thereby sidestepping numerous pitfalls:

- The app may have relationships such as has-many or belongs-to that are reflected in the table rows. Without knowing the details of these relationships, you might create an invalid subset of data. Using RottenPotatoes as an example, you might inadvertently end up with a `review` whose `movie_id` and `moviegoer_id` refer to nonexistent movies or moviegoers.
- Cloning the database eliminates possible differences in behavior between production and development resulting from differences in database implementations, difference in how certain data types such as dates are represented in different databases, and so on.
- Cloning gives you realistic valid data to work with in development.

If you can't clone the production database, or you have successfully cloned it but it's too unwieldy to use in development all the time, you can create a development database by extracting fixture data from the real database[5] using the steps in Figure 9.3.

Once the app is running in development, have one or two experienced customers demonstrate how they use the app, indicating during the demo what changes they have in mind (Nierstrasz et al. 2009). Ask them to talk through the demo as they go; although their comments will often be in terms of the user experience ("Now I'm adding Mona as an admin user"), if the app was created using BDD, the comments may reflect examples of the original user stories and therefore the app's architecture. Ask frequent questions during the demo, and if the maintainers of the app are available, have them observe the demo as well. In Section 9.3 we will see how these demos can form the basis of "ground truth" tests to underpin your changes.

Once you have an idea of how the app works, take a look at the database schema; Fred Brooks, Rob Pike, and others have all acknowledged the importance of understanding the data

https://gist.github.com/617ca988e1425a36dc84e6e973554d1c

```
# on production computer:
RAILS_ENV=production rake db:schema:dump
RAILS_ENV=production rake db:fixtures:extract
# copy db/schema.rb and test/fixtures/*.yml to development computer
# then, on development computer:
rake db:create         # uses RAILS_ENV=development by default
rake db:schema:load
rake db:fixtures:load
```

Figure 9.3: You can create an empty development database that has the same schema as the production database and then populate it with fixtures. Although Chapter 8 cautions against the abuse of fixtures, in this case we are using them to replicate known behavior from the production environment in your development environment.

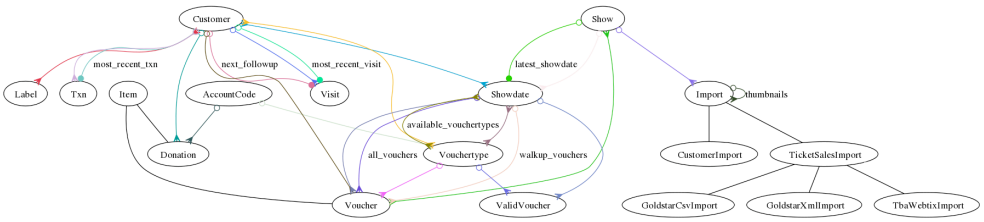

Figure 9.4: This simplified Unified Modeling Language (UML) class diagram, produced automatically by the `railroady` gem, shows the models in a Rails app that manages ticket sales, donations, and performance attendance for a small theater. Edges with arrowheads or circles show relationships between classes: a `Customer` has many `Visits` and `Vouchers` (open circle to arrowhead), has one `most_recent_visit` (solid circle to arrowhead), and a has and belongs to many `Labels` (arrowhead to arrowhead). Plain edges show inheritance: `Donation` and `Voucher` are subclasses of `Item`. (All of the important classes here inherit from `ActiveRecord::Base`, but `railroady` draws only the app's classes.) We will see other types of UML diagrams in Chapter 11.

structures as a key to understanding the app logic. You can use an interactive database GUI to explore the schema, but you might find it more efficient to run `rake db:schema:dump`, which creates a file `db/schema.rb` containing the database schema in the migrations DSL introduced in Section 4.2. The goal is to match up the schema with the app's overall architecture.

Figure 9.4 shows a simplified Unified Modeling Language (UML) class diagram generated by the `railroady` gem that captures the relationships among the most important classes and the most important attributes of those classes. While the diagram may look overwhelming initially, since not all classes play an equally important structural role, you can identify "highly connected" classes that are probably central to the application's functions. For example, in Figure 9.4, the `Customer` and `Voucher` classes are connected to each other and to many other classes. You can then identify the tables corresponding to these classes in the database schema.

Having familiarized yourself with the app's architecture, most important data structures, and major classes, you are ready to look at the code. The goal of inspecting the code is to get a sense of its overall quality, test coverage, and other statistics that serve as a proxy for how painful it may be to understand and modify. Therefore, before diving into any specific file, run `rake stats` to get the total number of lines of code and lines of tests for each file; this information can tell you which classes are most complex and therefore probably most important (highest LOC), best tested (best code-to-test ratio), simple "helper" classes (low LOC), and so on, deepening the understanding you bootstrapped from the class diagram and database schema. (Later in this chapter we'll show how to evaluate code with some

additional quality metrics to give you a heads up of where the hairiest efforts might be.) If test suites exist, run them; assuming most tests pass, read the tests to help understand the original developers' intentions. Then spend one hour (Nierstrasz et al. 2009) inspecting the code in the most important classes as well as those you believe you'll need to modify (the *change points*), which by now you should be getting a good sense of.

Summary of legacy code exploration:

- The goal of exploration is to understand how the app works from multiple stakeholders' points of view, including the customer requesting the changes and the designers and developers who created the original code.
- Exploration can be aided by reading tests, reading design documents if available, inspecting the code, and drawing or generating UML class diagrams to identify relationships among important entities (classes) in the app.
- Once you have successfully seen the app demonstrated in production, the next steps are to get it running in development by either cloning or fixturing the database and to get the test suite running in development.

■ ***Elaboration: Class–Responsibility–Collaborator (CRC) cards***

CRC cards (Figure 9.5) were proposed in 1989[6] as a way to help with object-oriented design. Each card identifies one class, its responsibilities, and collaborator classes with which it interacts to complete tasks. As this external screencast[7] shows, a team designing new code selects a user story (Section 7.1). For each story step, the team identifies or creates the CRC card(s) for the classes that participate in that step and confirms that the classes have the necessary Responsibilities and Collaborators to complete the step. If not, the collection of classes or responsibilities may be incomplete, or the division of responsibilities among classes may need to be changed. When exploring legacy code, you can create CRC cards to document the classes you find while following the flow from the controller action that handles a user story step through the models and views involved in the other story steps.

Self-Check 9.2.1. *What are some reasons it is important to get the app running in development even if you don't plan to make any code changes right away?*

◇ A few reasons include:

1. For SaaS, the existing tests may need access to a test database, which may not be accessible in production.
2. Part of your exploration might involve the use of an interactive debugger or other tools that could slow down execution, which would be disruptive on the live site.
3. For part of your exploration you might want to modify data in the database, which you can't do with live customer data.

■

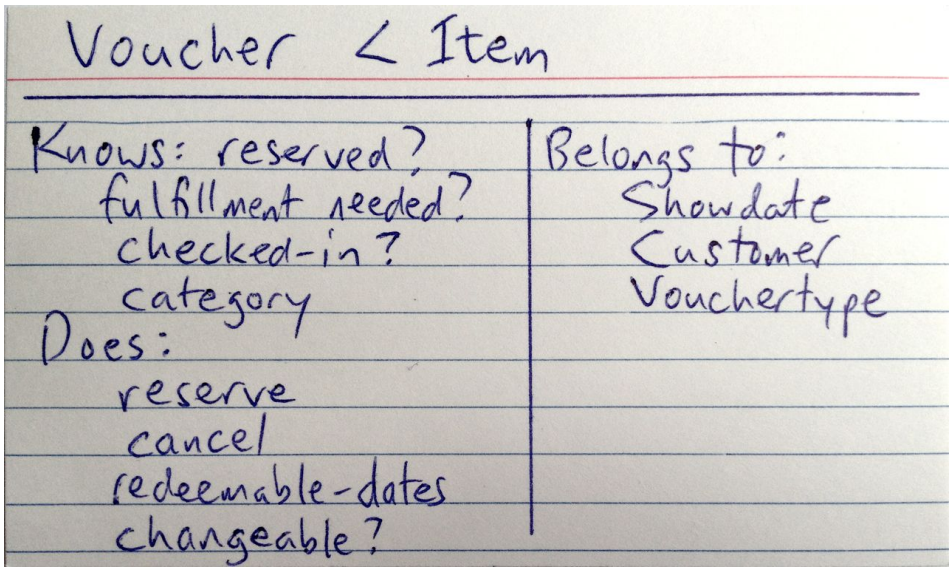

Figure 9.5: A 3-by-5 inch (or A7 size) Class–Responsibility–Collaborator (CRC) card representing the `Voucher` class from Figure 9.4. The left column represents `Voucher`'s responsibilities—things it knows (instance variables) or does (instance methods). Since Ruby instance variables are always accessed through instance methods, we can determine responsibilities by searching the class file `voucher.rb` for instance methods and calls to `attr_accessor`. The right column represents `Voucher`'s collaborator classes; for Rails apps we can determine many of these by looking for `has_many` and `belongs_to` in `voucher.rb`.

https://gist.github.com/b44d2059ebcccd4a1d45111884ba350b

```
# WARNING! This code has a bug! See text!
class TimeSetter
  def self.convert(d)
    y = 1980
    while (d > 365) do
      if (y % 400 == 0 ||
          (y % 4 == 0 && y % 100 != 0))
        if (d > 366)
          d -= 366
          y += 1
        end
      else
        d -= 365
        y += 1
      end
    end
    return y
  end
end
```

Figure 9.6: This method is hard to understand, hard to test, and therefore, by Feathers's definition of legacy code, hard to modify. In fact, it contains a bug—this example is a simplified version of a bug in the Microsoft Zune music player that caused any Zune booted on December 31, 2008, to freeze permanently, and for which the only resolution was to wait until the first minute of January 1, 2009, before rebooting.

9.3 Establishing Ground Truth With Characterization Tests

If there are no tests (or too few tests) covering the parts of the code affected by your planned changes, you'll need to create some tests. How do you do this given limited understanding of how the code works now? One way to start is to establish a baseline for "ground truth" by creating ***characterization tests***: tests written after the fact that capture and describe the *actual, current* behavior of a piece of software, even if that behavior has bugs. By creating a **R**epeatable automatic test (see Section 8.1) that mimics what the code does right now, you can ensure that those behaviors stay the same as you modify and enhance the code, like a high-level regression test.

It's often easiest to start with an integration-level characterization test such as a Cucumber scenario, since these make the fewest assumptions about how the app works and focus only on the user experience. Indeed, while good scenarios ultimately make use of a "domain language" rather than describing detailed user interactions in imperative steps (Section 7.8), at this point it's fine to start with imperative scenarios, since the goal is to increase coverage and provide ground truth from which to create more detailed tests. Once you have some green integration tests, you can turn your attention to unit- or functional-level tests, just as TDD follows BDD in the outside-in Agile cycle.

Whereas integration-level characterization tests just capture behaviors that we observe without requiring us to understand *how* those behaviors happen, a unit-level characterization test seems to require us to understand the implementation. For example, consider the code in Figure 9.6. As we'll discuss in detail in the next section, it has many problems, not least of which is that it contains a bug. The method `convert` calculates the current year given a starting year (in this case 1980) and the number of days elapsed since January 1 of that year. If 0 days have elapsed, then it is January 1, 1980; if 365 days have elapsed, it is December 31, 1980, since 1980 was a leap year; if 366 days have elapsed, it is January 1, 1981; and so on. How would we create unit tests for `convert` without understanding the method's logic in detail?

https://gist.github.com/ca621c822bd39c51e7d6aa27f783bbbe

```
require 'simplecov'
SimpleCov.start
require './time_setter'
describe TimeSetter do
  { 365 => 1980, 366 => 1981, 900 => 1982 }.each_pair do |arg,result|
    it "#{arg} days puts us in #{result}" do
      expect(TimeSetter.convert(arg)).to eq(result)
    end
  end
end
```

Figure 9.7: This simple spec, resulting from the reverse-engineering technique of creating characterization tests achieves 100% C0 coverage and helps us find a bug in Figure 9.6.

Feathers describes a useful technique for "reverse engineering" specs from a piece of code we don't yet understand: create a spec with an assertion that we know will probably fail, run the spec, and use the information in the error message to change the spec to match actual behavior. Essentially, we create specs that assert incorrect results, then fix the specs based on the actual test behavior. Our goal is to capture the current behavior as completely as possible so that we'll immediately know if code changes break the current behavior, so we aim for 100% C0 coverage (even though that's no guarantee of bug-freedom!), which is challenging because the code as presented has no seams. Doing this for `convert` results in the specs in Figure 9.7 and even finds a bug in the process!

Screencast 9.3.1: Creating characterization specs for TimeSetter.
`http://youtu.be/8QwvqtMp5QM`
We create specs that assert incorrect results, then fix them based on the actual test behavior. Our goal is to capture the current behavior as completely as possible so that we'll immediately know if code changes break the current behavior, so we aim for 100% C0 coverage (even though that's no guarantee of bug-freedom!), which is challenging because the code as presented has no seams. Our effort results in finding a bug that crippled thousands of Microsoft Zune players on December 31, 2008.

Summary of characterization tests:

- To Cover and Modify when we lack tests, we first create characterization tests that capture how the code works now.
- Integration-level characterization tests, such as Cucumber scenarios, are often easier to start with since they only capture externally visible app behavior.
- To create unit- and functional-level characterization tests for code we don't fully understand, we can write a spec that asserts an incorrect result, fix the assertion based on the error message, and repeat until we have sufficient coverage.

Self-Check 9.3.1. *State whether each of the following is a goal of unit and functional testing, a goal of characterization testing, or both:a*

i Improve coverage

https://gist.github.com/c350c1632f0e9fa7ca47c0c208cc9e8f

```
# Add one to i.
i += 1

# Lock to protect against concurrent access.
mutex = SpinLock.new

# This method swaps the panels.
def swap_panels(panel_1, panel_2)
  # ...
end
```

Figure 9.8: Examples of bad comments, which state the obvious. You'd be surprised how often comments just mimic code even in otherwise well-written apps. (Thanks to John Ousterhout for these examples and some of this advice on comments.)

ii Test boundary conditions and corner cases

iii Document intent and behavior of app code

iv Prevent regressions (reintroduction of earlier bugs)

◇ (i) and (iii) are goals of unit, functional, and characterization testing. (ii) and (iv) are goals of unit and functional testing, but non-goals of characterization testing. ■

■ ***Elaboration: What about specs that should pass, but don't?***

If the test suite is out-of-date, some tests may be failing red. Rather than trying to fix the tests before you understand the code, mark them as "pending" (for example, using RSpec's `pending` method) with a comment that reminds you to come back to them later to find out why they fail. Stick to the current task of preserving existing functionality while improving coverage, and don't get distracted trying to fix bugs along the way.

9.4 Comments and Commits: Documenting Code

Not only does legacy code often lack tests and good documentation, but its comments are often missing or inconsistent with the code. Thus far, we have not offered advice on how to write good comments, as we assume you already know how to write good code in this book. We now offer a brief sermon on comments, so that once you write successful characterization tests you can capture what you've learned by adding comments to the legacy code. Good comments have two properties:

1. They document things that aren't obvious from the code.
2. They are expressed at a higher level of abstraction than the code.

Figure 9.8 shows examples of comments that violate both properties, and Figure 9.9 shows a better example.

First, if you write comments as you code, much of what your code does is surely obvious to you, since you just wrote it. (Alas, *not* commenting as you go is a common defect of legacy code.) But if you or someone else reads your code later, long after you've forgotten those design ideas, comments should help you remember the non-obvious reasons you wrote

https://gist.github.com/f537017d1909733bbde757ea3abe951a

```
# Good Comment:
# Scan the array to see if the symbol exists

# Much better than:
# Loop through every array index, get the
# third value of the list in the content to
# determine if it has the symbol we are looking
# for. Set the result to the symbol if we
# find it.
```

Figure 9.9: Example of comments that raises the level of abstraction compared to comments that describe how you implement it.

the code the way you did. Examples of non-obvious things include the units for variables, code invariants, subtle problems that required a particular implementation, or unusual code that is there solely to work around some bug or account for a non-obvious boundary condition or corner case. In the case of legacy code, you are trying to add comments to document what went through another programmer's mind; once you figure it out, be sure to write it down before you forget!

Second, comments should raise the level of abstraction from the code. The programmer's goal is to write classes and other code that hides complexity; that is, to make it easier for others to use this existing code rather than re-create it themselves. Comments should therefore address concerns such as: What do I need to know to invoke this method? Are there preconditions, assumptions, or caveats? Among other jobs, a comment should provide enough of this information that someone who wants to call an existing class or method doesn't have to read its source code to figure these things out.

These guidelines are also generally true for commit messages, which you supply whenever you commit a set of code changes. However, one important principle is that you shouldn't put information in a commit message that a future developer will need to know while working on the code. Historical information—why a certain function was deleted or refactored, for example—is appropriate for including in a commit message. But information that a developer would need to know to use the code *as it exists now* should be in a comment, where the developer cannot fail to see it when they go to edit the code.

As with many other elements of Agile, when a process isn't working smoothly, it's trying to tell you something about your code. For example, we saw in Chapter 8 that when a test is hard to write due to the need for extensive mocking and stubbing, the test is trying to tell you that your code is not testable because it's poorly factored. Similarly here: if following the above guideline about comments vs. commits means you find yourself writing lots of cautionary caveats in the comments, your code is telling you that it might benefit from a refactoring cleanup so that you wouldn't need to post so many warning signs for the next developer who comes along with the intention of modifying it.

While virtually every other software engineering sermon in this book is paired with a tool that makes it easy for you to stay on the true path and for others to check if you have strayed, this is not the case for comments and commit messages. The only enforcement mechanism beyond self-discipline is inspection, which we discuss in Sections 10.4 and 10.7.

Summary of comments:

- Comments are best written at the same time as the code, not as an afterthought.
- Comments should not repeat what is obvious from the code. For example, explain *why* the code is written this way.
- Comments should raise the level of abstraction from the code, describing what a logical block of code does rather than providing line-by-line details.
- Commits should include historical information about why the code is the way it is, but information that developers need *while* using the current code belongs in comments. If this leads to too many comments, your code may need cleanup.

Self-Check 9.4.1. *True or False: One reason legacy code is long lasting is because it typically has good comments.*
◇ False. We wish it were true. Comments are often missing or inconsistent with the code, which is one reason it is called legacy code rather than beautiful code. ■

9.5 Metrics, Code Smells, and SOFA

> *7. Declining Quality - The quality of [software] systems will appear to be declining unless they are rigorously maintained and adapted to operational environment changes.*
>
> —Lehman's seventh law of software evolution

A key theme of this book is that engineering software is about creating not just working code, but *beautiful* working code. This chapter should make clear why we believe this: beautiful code is easier and less expensive to maintain. Given that software can live much longer than hardware, even engineers whose aesthetic sensibilities aren't moved by the idea of beautiful code can appreciate the practical economic advantage of reducing lifetime maintenance costs.

How can you tell when code is less than beautiful, and how do you improve it? We've all seen examples of code that's less than beautiful, even if we can't always pin down the specific problems. We can identify problems in two ways: quantitatively using ***software metrics*** and qualitatively using ***code smells***. Both are useful and tell us different things about the code, and we apply both to the ugly code in Figure 9.6.

Plan-and-Document
software projects sometimes include specific contractual requirements based on software metrics.

Software metrics are quantitative measurements of code complexity, which is often an estimate of the difficulty of thoroughly testing a piece of code. Dozens of metrics exist, and opinion varies widely on their usefulness, effectiveness, and "normal range" of values. Most metrics are based on the ***control flow graph*** of the program, in which each graph node represents a ***basic block*** (a set of statements that are always executed together), and an edge from node A to node B means that there is some code path in which B's basic block is executed immediately after A's.

Figure 9.10 shows the control flow graph corresponding to Figure 9.6, which we can use to compute two widely-used indicators of method-level complexity:

1. ***Cyclomatic complexity*** measures the number of linearly-independent paths through a piece of code.

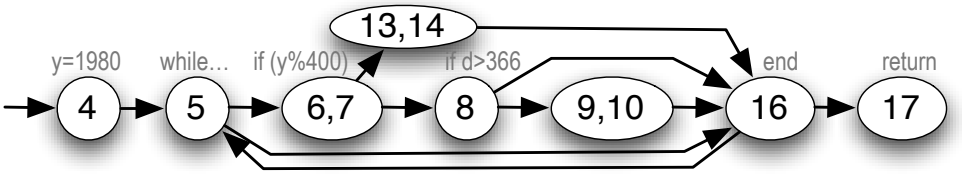

Figure 9.10: The node numbers in this control flow graph correspond to line numbers in Figure 9.6. Cyclomatic complexity is $E - N + 2P$ where E is the number of edges, N the number of nodes, and P the number of connected components. `convert` scores a cyclomatic complexity of 4 as measured by `saikuro` and an ABC score (Assignments, Branches, Conditionals) of 23 as measured by `flog`. Figure 9.11 puts these scores in context.

Metric	Tool	Target score	Book Reference
Code-to-test ratio	`rake stats`	$\leq 1:2$	Section 8.7
C0 coverage	`SimpleCov`	$\geq 90\%$	Section 8.7
ABC score	`flog` (`rake metrics`)	< 20/method	Section 9.5
Cyclomatic	`saikuro` (`rake metrics`)	< 10/method	Section 9.5

Figure 9.11: A summary of useful metrics we've seen so far that highlight the connection between beauty and testability, including Ruby tools that compute them and suggested "normal" ranges. (The recommended value for cyclomatic complexity comes from NIST, the U.S. National Institute of Standards and Technologies.) The `metric_fu` gem includes `flog`, `saikuro`, and additional tools for computing metrics we'll meet in Chapter 11.

Software engineer Frank McCabe Sr. invented the cyclomatic complexity metric in 1976.

2. ***ABC score*** is a weighted sum of the number of **A**ssignments, **B**ranches and **C**onditionals in a piece of code.

These analyses are usually performed on source code and were originally developed for statically-typed languages. In dynamic languages, the analyses are complicated by metaprogramming and other mechanisms that may cause changes to the control flow graph at runtime. Nonetheless, they are useful first-order metrics, and as you might expect, the Ruby community has developed tools to measure them. `saikuro` computes a simplified version of cyclomatic complexity and `flog` computes a variant of the ABC score that is weighted in a way appropriate for Ruby idioms. Both of these and more are included in the `metric_fu` gem (part of the courseware). Running `rake metrics` on a Rails app computes various metrics including these, and highlights parts of the code in which multiple metrics are outside their recommended ranges. In addition, CodeClimate[8] provides many of these metrics as a service: by creating an account there and linking your GitHub repository to it, you can view a "report card" of your code metrics anytime, and the report is automatically updated when you push new code to GitHub. Figure 9.11 summarizes useful metrics we've seen so far that speak to testability and therefore to code beauty.

The second way to spot code problems is by looking for ***code smells***, which are structural characteristics of source code not readily captured by metrics. Like real smells, code smells call our attention to places that *may* be problematic. Martin Fowler's classic book on refactoring (Fowler et al. 1999) lists 22 code smells, four of which we show in Figure 9.12, and Robert C. Martin's *Clean Code* (Martin 2008) has one of the more comprehensive catalogs with an amazing 63 code smells, of which three are specific to Java, nine are about testing, and the remainder are more general.

Design smells (see Chapter 11) tell us when something's wrong in the way classes interact, rather than within the methods of a specific class.

Four particular smells that appear in Martin's *Clean Code* are worth emphasizing, because they are symptoms of other problems that you can often fix by simple refactorings. These four are identified by the acronym **SOFA**, which states that a well-written method should:

Name	Symptom	Possible refactorings
Shotgun Surgery	Making a small change to a class or method results in lots of little changes rippling to other classes or methods.	Use Move Method or Move Field to bring all the data or behaviors into a single place.
Data Clump	The same three or four data items seem to often be passed as arguments together or manipulated together.	Use Extract Class or Preserve Whole Object to create a class that groups the data together, and pass around instances of that class.
Inappropriate Intimacy	One class exploits too much knowledge about the implementation (methods or attributes) of another.	Use Move Method or Move Field if the methods really need to be somewhere else, use Extract Class if there is true overlap between two classes, or introduce a Delegate to hide the implementation.
Repetitive Boilerplate	You have bits of code that are the same or nearly the same in various different places (non-DRY).	Use Extract Method to pull redundant code into its own method that the repetitive places can call. In Ruby, you can even use `yield` to extract the "enclosing" code and having it yield back to the non-repetitive code.

Figure 9.12: Four whimsically-named code smells from Fowler's list of 22, along with the refactorings (some of which we'll meet in the next section) that might remedy the smell if applied. Refer to Fowler's book for the refactorings mentioned in the table but not introduced in this book.

- be **S**hort, so that its main purpose is quickly grasped;
- do only **O**ne thing, so testing can focus on thoroughly exercising that one thing;
- take **F**ew arguments, so that all-important combinations of argument values can be tested;
- maintain a consistent level of **A**bstraction, so that it doesn't jump back and forth between saying *what to do* and saying *how to do it*.

Figure 9.6 violates at least the first and last of these, and exhibits other smells as well, as we can see by running `reek` on it:

https://gist.github.com/223f7e7b28b390b650dd196f80048a2f

```
1 time_setter.rb -- 5 warnings:
2  TimeSetter#self.convert calls (y + 1) twice (Duplication)
3  TimeSetter#self.convert has approx 6 statements (LongMethod)
4  TimeSetter#self.convert has the parameter name 'd' (UncommunicativeName)
5  TimeSetter#self.convert has the variable name 'd' (UncommunicativeName)
6  TimeSetter#self.convert has the variable name 'y' (UncommunicativeName)
```

Not DRY (line 2). Admittedly this is only a minor duplication, but as with any smell, it's worth asking ourselves why the code turned out that way.

Uncommunicative names (lines 4–6). Variable `y` appears to be an integer (lines 6, 7, 10, 14) and is related to another variable `d`—what could those be? For that matter, what does the class `TimeSetter` set the time to, and what is being converted to what in `convert`? Four decades ago, memory was precious and so variable names were kept short to allow more space for code. Today, there's no excuse for poor variable names; Figure 9.13 provides suggestions.

Too long (line 3). More lines of code per method means more places for bugs to hide, more paths to test, and more mocking and stubbing during testing. However, excessive length is really a symptom that emerges from more specific problems—in this case, failure to stick

What	Guideline	Example
Variable or class name	Noun phrase	`PopularMovie`, `top_movies`
Method with side effects	Verb phrase	`pay_for_order`, `charge_credit_card!`
Method that returns a value	Noun phrase	`movie.producers`, `actor_list`
Boolean variable or method	Adjective phrase	`already_rated?`, `@is_oscar_winner`

Figure 9.13: variable-naming guidelines based on simple English, excerpted from Green and Ledgard 2011. Given that disk space is free and modern editors have auto-completion that saves you retyping the full name, your colleagues will thank you for writing `@is_oscar_winner` instead of `OsWin`.

https://gist.github.com/2183dec77d51a632f93923c6c3946fe6

```
1 start with Year = 1980
2 while (days remaining > 365)
3   if Year is a leap year
4     then if possible, peel off 366 days and advance Year by 1
5   else
6     peel off 365 days and advance Year by 1
7 return Year
```

Figure 9.14: The computation of the current year given the number of days since the beginning of a start year (1980) is much more clear when written in pseudocode. Notice that *what the method does* is quick to grasp, even though each step would have to be broken down into more detail when turned into code. We will refactor the Ruby code to match the clarity and conciseness of this pseudocode.

to a single level of **A**bstraction. As Figure 9.14 shows, `convert` really consists of a small number of high-level steps, each of which could be divided into sub-steps. But in the code, there is no way to tell where the boundaries of steps or sub-steps would be, making the method harder to understand. Indeed, the nested conditional in lines 6–8 makes it hard for a programmer to mentally "walk through" the code, and complicates testing since you have to select sets of test cases that exercise each possible code path.

The ancient wisdom that a method shouldn't exceed one screenful of code was based on text-only terminals with 24 lines of 80 characters. A modern 22-inch monitor shows 10 times that much, so guidelines like SOFA are more reliable today.

As a result of these deficiencies, you probably had to work hard to figure out what this relatively simple method does. (You might blame this on a lack of comments in the code, but once the above smells are fixed, there will be hardly any need for them.) Astute readers usually note the constants 1980, 365, and 366, and infer that the method has something to do with leap years and that 1980 is special. In fact, `convert` calculates the current year given a starting year of 1980 and the number of days elapsed since January 1 of that year, as Figure 9.14 shows using simple pseudocode. In Section 9.5, we will make the Ruby code as transparent as the pseudocode by ***refactoring*** it—applying transformations that improve its structure without changing its behavior.

A few specific examples of doing one thing are worth calling out because they occur frequently:

- Handling an exception is one thing. If method M computes something and also tries to handle various exceptions that could arise while doing so, consider splitting out a method M' that just does the work, and having M do exception handling and otherwise delegate the work to M'.
- Queries (computing something) and commands (doing something that causes a side effect) are distinct, so a method should either compute something that is side-effect-free or it should cause a specific side effect, but not both. Such violations of *command–query separation* also complicate testing.

Summary

- Software metrics provide a quantitative measure of code quality. While opinion varies on which metrics are most useful and what their "normal" values should be (especially in dynamic languages such as Ruby), metrics such as cyclomatic complexity and ABC score can be used to guide your search toward code that is in particular need of attention, just as low C0 coverage identifies undertested code.
- Code smells provide qualitative but specific descriptions of problems that make code hard to read. Depending on which catalog you use, over 60 specific code smells have been identified.
- The acronym SOFA names four desirable properties of a method: it should be **S**hort, do **O**ne thing, have **F**ew arguments, and maintain a single level of **A**bstraction.

Self-Check 9.5.1. *Give an example of a dynamic language feature in Ruby that could distort metrics such as cyclomatic complexity or ABC score.*

◇ Any metaprogramming mechanism could do this. A trivial example is `s="if (d>=366)[...]"; eval s`, since the evaluation of the string would cause a conditional to be executed even though there's no conditional in the code itself, which contains only an assignment to a variable and a call to the `eval` method. A subtler example is a method such as `before_filter` (Section 5.1), which essentially adds a new method to a list of methods to be called before a controller action. ■

Self-Check 9.5.2. *Which SOFA guideline—be **S**hort, do **O**ne thing, have **F**ew arguments, stick to a single level of **A**bstraction—do you think is most important from a unit-testability point of view?*

◇ **F**ew arguments implies fewer ways that code paths in the method can depend on the arguments, making testing more tractable. **S**hort methods are certainly easier to test, but this property usually follows when the other three are observed. ■

9.6 Method-Level Refactoring: Replacing Dependencies With Seams

> *2. Increasing Complexity - As [a software] system evolves, its complexity increases unless work is done to maintain or reduce it.*
>
> —Lehman's second law of software evolution

With the characterization specs developed in Section 9.3, we have a solid foundation on which to base our refactoring to repair the problems identified in Section 9.5. The term *refactoring* refers not only to a general process, but also to an instance of a specific code transformation. Thus, just as with code smells, we speak of a catalog of refactorings, and there are many such catalogs to choose from. We prefer Fowler's catalog, so the examples in this chapter follow Fowler's terminology and are cross-referenced to Chapters 6, 8, 9, and 10 of his book *Refactoring: Ruby Edition* (Fields et al. 2009). While the correspondence between code smells and refactorings is not perfect, in general each of those chapters describes a group of method-level refactorings that address specific code smells or problems,

Name (Chapter)	Problem	Solution
Extract method (6)	You have a code fragment that can be grouped together.	Turn the fragment into a method whose name explains the purpose of the method.
Decompose Conditional (9)	You have a complicated conditional (if-then-else) statement.	Extract methods from the condition, "then" part, and "else" part(s).
Replace Method with Method Object (6)	You have a long method that uses local variables in such a way that you cannot apply Extract Method.	Turn the method into its own object so that all the local variables become instance variables on that object. You can then decompose the method into other methods on the same object.
Replace Magic Number with Symbolic Constant (8)	You have a literal number with a particular meaning.	Create a constant, name it after the meaning, and replace the number with it.

Figure 9.15: Four example refactorings, with parentheses around the chapter in which each appears in Fowler's book. Each refactoring has a name, a problem that it solves, and an overview of the code transformation(s) that solve the problem. Fowler's book also includes detailed mechanics for each refactoring, as Figure 9.16 shows.

and further chapters describe refactorings that affect multiple classes, which we'll learn about in Chapter 11.

Each refactoring consists of a descriptive name and a step-by-step process for transforming the code via small incremental steps, testing after each step. Most refactorings will cause at least temporary test failures, since unit tests usually depend on implementation, which is exactly what refactoring changes. A key goal of the refactoring process is to minimize the amount of time that tests are failing (red); the idea is that each refactoring step is small enough that adjusting the tests to pass before moving on to the next step is not difficult. If you find that getting from red back to green is harder than expected, you must determine if your understanding of the code was incomplete, or if you have really broken something while refactoring.

Getting started with refactoring can seem overwhelming: without knowing what refactorings exist, it may be hard to decide how to improve a piece of code. Until you have some experience improving pieces of code, it may be hard to understand the explanations of the refactorings or the motivations for when to use them. Don't be discouraged by this apparent chicken-and-egg problem; like TDD and BDD, what seems overwhelming at first can quickly become familiar.

As a start, Figure 9.15 shows four of Fowler's refactorings that we will apply to our code. In his book, each refactoring is accompanied by an example and an extremely detailed list of mechanical steps for performing the refactoring, in some cases referring to other refactorings that may be necessary in order to apply this one. For example, Figure 9.16 shows the first few steps for applying the Extract Method refactoring. With these examples in mind, we can refactor Figure 9.6.

Long method is the most obvious code smell in Figure 9.6, but that's just an overall symptom to which various specific problems contribute. The high ABC score (23) of `convert` suggests one place to start focusing our attention: the condition of the `if` in lines 6–7 is difficult to understand, and the conditional is nested two-deep. As Figure 9.15 suggests, a hard-to-read conditional expression can be improved by applying the very common refactoring *Decompose Conditional*, which in turn relies on *Extract Method*. We move some code

1. Create a new method, and name it after the intention of the method (name it by what it does, not by how it does it). If the code you want to extract is very simple, such as a single message or function call, you should extract it if the name of the new method reveals the intention of the code in a better way. If you can't come up with a more meaningful name, don't extract the code.
2. Copy the extracted code from the source method into the new target method.
3. Scan the extracted code for references to any variables that are local in scope to the source method. These are local variables and parameters to the method.
4. See whether any temporary variables are used only within this extracted code. If so, declare them in the target method as temporary variables.
5. Look to see whether any of these local-scope variables are modified by the extracted code. If one variable is modified, see whether you can treat the extracted code as a query and assign the result to the variable concerned. If this is awkward, or if there is more than one such variable, you can't extract the method as it stands. You may need to use *Split Temporary Variable* and try again. You can eliminate temporary variables with *Replace Temp with Query* (see the discussion in the examples).
6. Pass into the target method as parameters local-scope variables that are read from the extracted method.
7. ...

Figure 9.16: Fowler's detailed steps for the Extract Method refactoring. In his book, each refactoring is described as a step-by-step code transformation process that may refer to other refactorings.

into a new method with a descriptive name, as Figure 9.17 shows. Note that in addition to making the conditional more readable, the separate definition of `leap_year?` makes the leap year calculation separately testable and provides a seam at line 6 where we could stub the method to simplify testing of `convert`, similar to the example in the Elaboration at the end of Section 8.4. In general, when a method mixes code that says *what to do* with code that says *how to do it*, this may be a warning to check whether you need to use Extract Method in order to maintain a consistent level of **A**bstraction.

The conditional is also nested two-deep, making it hard to understand and increasing `convert`'s ABC score. The *Decompose Conditional* refactoring also breaks up the complex condition by replacing each arm of the conditional with an extracted method. Notice, though, that the two arms of the conditional correspond to lines 4 and 6 of the pseudocode in Figure 9.14, both of which have the *side effects* of changing the values of `d` and `y` (hence our use of `!` in the names of the extracted methods). In order for those side effects to be visible to `convert`, we must turn the local variables into class variables throughout `TimeSetter`, giving them more descriptive names `@@year` and `@@days_remaining` while we're at it. Finally, since `@@year` is now a class variable, we no longer need to pass it as an explicit argument to `leap_year?`. Figure 9.18 shows the result.

As long as we're cleaning up, the code in Figure 9.18 also fixes two minor code smells. The first is uncommunicative variable names: `convert` doesn't describe very well what this method does, and the parameter name `d` is not useful. The other is the use of "magic number" literal constants such as 1980 in line 4; we apply *Replace Magic Number with Symbolic Constant* (Fowler chapter 8) to replace it with the more descriptive constant name `STARTING_YEAR`. What about the other constants such as 365 and 366? In this example, they're probably familiar enough to most programmers to leave as-is, but if you saw 351 rather than 365, and if line 26 (in `leap_year?`) used the constant 19 rather than 4, you might not recognize the leap year calculation for the ***Hebrew calendar***. Remember that refactoring only improves the code for human readers; the computer doesn't care. So in such

https://gist.github.com/44ef9cea09c14d79a37ef43f66168cf7

```
1  # NOTE: line 7 fixes bug in original version
2  class TimeSetter
3    def self.convert(d)
4      y = 1980
5      while (d > 365) do
6        if leap_year?(y)
7          if (d >= 366)
8            d -= 366
9            y += 1
10         end
11       else
12         d -= 365
13         y += 1
14       end
15     end
16     return y
17   end
18   private
19   def self.leap_year?(year)
20     year % 400 == 0 ||
21       (year % 4 == 0 && year % 100 != 0)
22   end
23 end
```

Figure 9.17: Applying the Extract Method refactoring to lines 3–4 of Figure 9.6 makes the conditional's purpose immediately clear (line 6) by replacing the condition with a well-named method (lines 19–22), which we declare `private` to keep the class's implementation details well encapsulated. For even more transparency, we could apply Extract Method again to `leap_year?` by extracting methods `every_400_years?` and `every_4_years_except_centuries?`.

https://gist.github.com/35461c627212c8098f6517855331ab02

```
1  # NOTE: line 7 fixes bug in original version
2  class TimeSetter
3    ORIGIN_YEAR = 1980
4    def self.calculate_current_year(days_since_origin)
5      @@year = ORIGIN_YEAR
6      @@days_remaining = days_since_origin
7      while (@@days_remaining > 365) do
8        if leap_year?
9          peel_off_leap_year!
10       else
11         peel_off_regular_year!
12       end
13     end
14     return @@year
15   end
16   private
17   def self.peel_off_leap_year!
18     if (@@days_remaining >= 366)
19       @@days_remaining -= 366 ; @@year += 1
20     end
21   end
22   def self.peel_off_regular_year!
23     @@days_remaining -= 365 ; @@year += 1
24   end
25   def self.leap_year?
26     @@year % 400 == 0 ||
27       (@@year % 4 == 0 && @@year % 100 != 0)
28   end
29 end
```

Figure 9.18: We decompose the conditional in line 7 by replacing each branch with an extracted method. Note that while the total number of lines of code has increased, `convert` itself has become Shorter, and its steps now correspond closely to the pseudocode in Figure 9.14, sticking to a single level of Abstraction while delegating details to the extracted helper methods.

https://gist.github.com/e4028ce92d0109b232142a45c3d03c66

```
# An example call would now be:
#  year = TimeSetter.new(367).calculate_current_year
# rather than:
#  year = TimeSetter.calculate_current_year(367)
class TimeSetter
  ORIGIN_YEAR = 1980
  def initialize(days_since_origin)
    @year = ORIGIN_YEAR
    @days_remaining = days_since_origin
  end
  def calculate_current_year
    while (@days_remaining > 365) do
      if leap_year?
        peel_off_leap_year!
      else
        peel_off_regular_year!
      end
    end
    return @year
  end
  private
  def peel_off_leap_year!
    if (@days_remaining >= 366)
      @days_remaining -= 366 ; @year += 1
    end
  end
  def peel_off_regular_year!
    @days_remaining -= 365 ; @year += 1
  end
  def leap_year?
    @year % 400 == 0 ||
      (@year % 4 == 0 && @year % 100 != 0)
  end
end
```

Figure 9.19: If we use Fowler's recommended refactoring, the code is cleaner because we now use instance variables rather than class variables to track side effects, but it changes the way `calculate_current_year` is called because it's now an instance method. This would break existing code and tests, and so might be deferred until later in the refactoring process.

cases use your judgment as to how much refactoring is enough.

In our case, re-running `flog` on the refactored code in Figure 9.18 brings the ABC score for the newly-renamed `calculate_current_year` from 23.0 down to 6.6, which is well below the suggested NIST threshold of 10.0. Also, `reek` now reports only two smells. The first is "low cohesion" for the helper methods `peel_off_leap_year` and `peel_off_regular_year`; this is a design smell, and we will discuss what it means in Chapter 11. The second smell is declaration of class variables inside a method. When we applied Decompose Conditional and Extract Method, we turned local variables into class variables `@@year` and `@@days_remaining` so that the newly-extracted methods could successfully modify those variables' values. Our solution is effective, but clumsier than *Replace Method with Method Object* (Fowler chapter 6). In that refactoring, the original method `convert` is turned into an object *instance* (rather than a class) whose instance variables capture the object's state; the helper methods then operate on the instance variables.

Figure 9.19 shows the result of applying such a refactoring, but there is an important caveat. So far, none of our refactorings have caused our characterization specs to fail, since the specs were just calling `TimeSetter.convert`. But applying *Replace Method With Method Object* changes the calling interface to `convert` in a way that makes tests fail. If we were working with real legacy code, we would have to find every site that calls `convert`,

change it to use the new calling interface, and change any failing tests accordingly. In a real project, we'd want to avoid changes that needlessly break the calling interface, so we'd need to consider carefully whether the readability gained by applying this refactoring would outweigh the risk of introducing this breaking change.

Summary of refactoring:

- A refactoring is a particular transformation of a piece of code, including a name, description of when to use the refactoring and what it does, and detailed sequence of mechanical steps to apply the refactoring. Effective refactorings should improve software metrics, eliminate code smells, or both.
- Although most refactorings will inevitably cause some existing tests to fail (if not, the code in question is probably undertested), a key goal of the refactoring process is to minimize the amount of time until those tests are modified and once again passing green.
- Sometimes applying a refactoring may result in recursively having to apply simpler refactorings first, as *Decompose Conditional* may require applying *Extract Method*.

■ ***Elaboration: Refactoring and language choice***

Some refactorings compensate for programming language features that may encourage bad code. For example, one suggested refactoring for adding seams is *Encapsulate Field*, in which direct access to an object's instance variables is replaced by calls to getter and setter methods. This makes sense in Java, but as we've seen, getter and setter methods provide the *only* access to a Ruby object's instance variables from outside the object. (The refactoring still makes sense inside the object's own methods, as the Elaboration at the end of Section 2.3 suggests.) Similarly, the *Generalize Type* refactoring suggests creating more general types to improve code sharing, but Ruby's mixins and duck typing make such sharing easy. As we'll see in Chapter 11, it's also the case that some design patterns are simply unnecessary in Ruby because the problem they solve doesn't arise in dynamic languages.

Self-Check 9.6.1. *Which is* not *a goal of method-level refactoring: (a) reducing code complexity, (b) eliminating code smells, (c) eliminating bugs, (d) improving testability?*
◇ (c). While debugging is important, the goal of refactoring is to *preserve* the code's current behavior while changing its structure. ■

9.7 The Plan-And-Document Perspective on Working With Legacy Code

One reason for the term ***lifecycle*** from Chapter 1 is that a software product enters a maintenance phase after development completes. Roughly two-thirds of the costs are in maintenance versus one-third in development. One reason that companies charge roughly 10% of the price of software for annual maintenance is to pay the team that does the maintenance.

Organizations following Plan-And-Document processes typically have different teams for development and maintenance, with developers being redistributed onto new projects once the project is released. Thus, we now have a *maintenance manager* who takes over the roles of the project manager during development, and we have *maintenance software engineers*

working on the team that make the changes to the code. Sadly, maintenance engineering has an unglamorous reputation, so it is typically performed by either the newest or least accomplished managers and engineers in an organization. Many organizations use different people for Quality Assessment to do the testing and for user documentation.

For software products developed using Plan-And-Document processes, the environment for maintenance is very different from the environment for development:

- *Working software*—A working software product is in the field during this whole phase, and new releases must not interfere with existing features.
- *Customer collaboration*—Rather than trying to meet a specification that is part of a negotiated contract, the goal for this phase is to work with customers to improve the product for the next release.
- *Responding to change*—Based on use of the product, customers send a stream of ***change requests***, which can be new features as well as bug fixes. One challenge of the maintenance phase is prioritizing whether to implement a change request and in which release should it appear.

Change requests are called *maintenance requests* in IEEE standards.

Regression testing plays a much bigger role in maintenance to avoid breaking old features when developing new ones. Refactoring also plays a much bigger role, as you may need to refactor to implement a change request or simply to make the code more maintainable. There is less incentive for the extra cost and time to make the product easier to maintain in Plan-And-Document processes initially if the company developing the software is not the one that maintains it, which is one reason refactoring plays a smaller role during development.

As mentioned above, ***change management*** is based on change requests made by customers and other stakeholders to fix bugs or to improve functionality (see Section 10.7). They typically fill out ***change request forms***, which are tracked using a ticket tracking system so that each request is responded to and resolved. A key tool for change management is a version control system, which tracks all modifications to all objects, as we describe in Sections 10.3 and 10.2.

The prior paragraphs should sound familiar, for we are describing Agile development; in fact, the three bullets are copied from the Agile Manifesto (see Section 1.3). Thus, *maintenance is essentially an Agile process*. Change requests are like user stories; the triaging of change requests is similar to the assignment of points and using Pivotal Tracker to decide how to prioritize stories; and new releases of the software product act as Agile iterations of the working prototype. Plan-and-document maintenance even follows the same strategy of breaking a large change request into many smaller ones to make them easier to assess and implement, just as we do with user stories assigned more than eight points (see Section 7.4). Hence, if the same team is developing and maintaining the software, nothing changes after the first release of the product when using the Agile lifecycle.

Although one paper reports successfully using an Agile process to maintain software developed using Plan-And-Document processes (Poole and Huisman 2001), normally an organization that follows Plan-And-Document for development also follows it for maintenance. As we saw in earlier chapters, this process expects a strong project manager who makes the cost estimate, develops the schedule, reduces risks to the project, and formulates a careful plan for all the pieces of the project. This plan is reflected in many documents, which we saw in Figures 7.6 and 8.14 and will see in the next chapter in Figures 10.11, 10.12, and 10.13. Thus, the impact of change in Plan-And-Document processes is not just the cost to change the

Tasks	*In Plan and Document*	*In Agile*
Customer change request	Change request forms	User story on 3x5 cards in Connextra format
Change request cost/time estimate	By Maintenance Manager	Points by Development Team
Triage of change requests	Change Control Board	Development team with customer participation
Roles	Maintenance Manager	N.A.
	Maintenance SW Engineers QA team Documentation teams Customer support group	Development team

Figure 9.20: The relationship between the maintenance related tasks of Plan-and-Document versus Agile methodologies.

code, but also to change the documentation and testing plan. Given the many more objects of Plan-And-Document, it takes more effort to synchronize to keep them all consistent when a change is made.

A *change control board* examines all significant requests to decide if the changes should be included in the next version of the system. This group needs estimates of the cost of a change to decide whether or not to approve the change request. The maintenance manager must estimate the effort and time to implement each change, much as the project manager did for the project initially (see Section 7.9). The group also asks the QA team for the cost of testing, including running all the regression tests and developing new ones (if needed) for a change. The documentation group also estimates the cost to change the documentation. Finally, the customer support group checks whether there is a workaround to decide if the change is urgent or not. Besides cost, the group considers the increased value of the product after the change when deciding what to do.

To help keep track what must be done in Plan-And-Document processes, you will not be surprised to learn that IEEE offers standards to help. Figure 9.21 shows the outline of a maintenance plan from the IEEE Maintenance Standard 1219-1998.

Ideally, changes can all be scheduled to keep the code, documents, and plans all in synchronization with an upcoming release. Alas, some changes are so urgent that everything else is dropped to try to get the new version to the customer as fast as possible. For example:

- The software product crashes.
- A security hole has been identified that makes the data collected by the product particularly vulnerable.
- New releases of the underlying operating system or libraries force changes to the product for it to continue to function.
- A competitor brings out product or feature that if not matched will dramatically affect the business of the customer.
- New laws are passed that affect the product.

While the assumption is that the team will update the documentation and plans as soon as the emergency is over, in practice emergencies can be so frequent that the maintenance

Table of Contents

1. Introduction
2. References
3. Definitions
4. Software Maintenance Overview
   4.1 Organization
   4.2 Scheduling Priorities
   4.3 Resource Summary
   4.4 Responsibilities
   4.5 Tools, Techniques, and Methods
5. Software Maintenance Process
   5.1 Problem/modification identification/classification, and prioritization
   5.2 Analysis
   5.3 Design
   5.4 Implementation
   5.5 System Testing
   5.6 Acceptance Testing
   5.7 Delivery
6. Software Maintenance Reporting Requirements
7. Software Maintenance Administrative Requirements
   7.1 Anomaly Resolution and Reporting
   7.2 Deviation Policy
   7.3 Control Procedures
   7.4 Standards, Practices, and Conventions
   7.5 Performance Tracking
   7.6 Quality Control of Plan
8. Software Maintenance Documentation Requirements

Figure 9.21: Maintenance plan outline from the IEEE 1219-1998 Standard for Maintenance in Systems and Software Engineering.

team can't keep everything in synch. Such a buildup is called a ***technical debt***. Such procrastination can lead to code that is increasingly difficult to maintain, which in turn leads to an increasing need to refactor the code as the code's "viscosity" makes it more and more difficult to add functionality cleanly. While refactoring is a natural part of Agile, it less likely for the Change Control Committee to approve changes that require refactoring, as these changes are much more expensive. That is–as the name is intended to indicate–if you don't repay your technical debt, it grows: the "uglier" the code gets, the more error-prone and time-consuming it is to refactor!

Backfilling is the term maintenance engineers use to describe getting code back in synch after emergencies.

In addition to estimating the cost of each potential change for the Change Control Board, an organization's management may ask what will be the annual cost of maintenance of a project. The maintenance manager may base this estimate on software metrics, just as the project manager may use metrics to estimate the cost to develop a project (see Section 7.9). The metrics are different for maintenance, as they are measuring the maintenance process. Examples of metrics that may indicate increased difficulty of maintenance include the average time to analyze or implement a change request and increases in the number of change requests made or approved.

At some point in the lifecycle of a software product, the question arises whether it is time for it to be replaced. An alternative that is related to refactoring is called *reengineering*. Like refactoring, the idea is to keep functionality the same but to make the code much easier to maintain. Examples include:

- Changing the database schema.
- Using a reverse engineering tool to improve documentation.
- Using a structural analysis tool to identify and simplify complex control structures.
- Using a language translation tool to change code from a procedure-oriented language like C or COBOL to an object-oriented language like C++ or Java.

The hope is that reengineering will be much less expensive and much more likely to succeed than reimplementing the software product from scratch.

Summary: The insight from this section is that you can think of Agile as a maintenance process, in that change is the norm, you are in continuous contact with the customer, and that new iterations of the product are routinely deployed to the customer as new releases. Hence, regression testing and refactoring are standard in the Agile process just as they are the maintenance phase of Plan-and-Document. In Plan-and-Document processes:

- *Maintenance managers* play the role of project managers: they interface with the customer and upper management, make the cost and schedule estimates, documents the maintenance plan, and manage the *maintenance software engineers*.
- Customers and other stakeholders issue ***change requests***, which a *Change Control Committee* triages based on the benefit of the change and cost estimates from the maintenance manager, the documentation team, and the QA team.
- ***Regression testing*** plays a bigger role in maintenance to ensure that new features do not interfere with old ones.
- ***Refactoring*** plays a bigger role as well, in part because there is often less refactoring in Plan-and-Document processes during product development than in Agile development.
- An alternative to starting over when the code becomes increasingly difficult to maintain is to *reengineer* the code to lower the cost of having a much more maintainable system.

One argument for Agile development is therefore as follows: if two-thirds of the cost of product are in the maintenance phase, why not use the same maintenance-compatible software development process for the whole lifecycle?

Self-Check 9.7.1. *True or False: The cost of maintenance usually exceeds the cost of development.*
◇ True. ■

Self-Check 9.7.2. *True or False: Refactoring and reengineering are synonyms.*
◇ False: While related terms, reengineering often relies on automatic tools and occurs as software ages and maintainability becomes more difficult, yet refactoring is a continuous process of code improvement that happens during both development and maintenance. ■

Self-Check 9.7.3. *Match the Plan-and-Document maintenance terms on the left to the Agile terms on the right:*

Change request	*Iteration*
Change request cost estimate	*Icebox, Active columns in Pivotal Tracker*
Change request triage	*Points*
Release	*User story*

◇ Change request $\iff$ User story; Change request cost estimate $\iff$ Points; Release $\iff$ Iteration; and Change request triage $\iff$ Icebox, Active columns in Pivotal Tracker. ■

9.8 Fallacies and Pitfalls

Pitfall: **Using TDD and CRC to think only tactically and not strategically about design.**

The extreme version of CRC cards seems to fit well with Agile: design and build the simplest thing that could possibly work, and embrace the fact that you'll need to change it later. But it's possible to take this approach too far. One suggestion from accomplished software craftsman and engineer John Ousterhout is to "design it twice": use CRC cards to come up with a design, then put it aside and try a different design from scratch, perhaps thinking a bit adversarially about how you want to beat the team that did the original design. If you're unable to improve on the original design, you can be more confident that it represents a reasonable starting point. But surprisingly often, you'll find a simpler or more elegant design after you've had a chance to think through the problem the first time.

Pitfall: **Conjoined Methods**

Ousterhout also advises Ousterhout 2018 warns against creating *conjoined methods*: two methods that collaborate tightly in accomplishing one goal, so that there is a lot of interaction between them and neither can be effectively understood without also understanding the other. This advice is consistent with the SOFA advice that a method should do **O**ne thing (Ousterhout would say that each of the conjoined methods only does part of a thing) but is an easy pitfall to experience if you're overzealous in making methods **S**hort. One sign of this is that it's nearly impossible to isolate one method from the other in tests; this is different from a helper method, which breaks out a well-defined subtask that can be individually tested.

Pitfall: **Conflating refactoring with enhancement.**

When you're refactoring or creating additional tests (such as characterization tests) in preparation to improve legacy code, there is a great temptation to fix "little things" along the way: methods that look just a little messy, instance variables that look obsolete, dead code that looks like it's never reached from anywhere, "really simple" features that look like you could quickly add while doing other tasks. *Resist these temptations!* First, the reason to establish ground-truth tests ahead of time is to bootstrap yourself into a position from which you can make changes with confidence that you're not breaking anything. Trying to make such "improvements" in the absence of good test coverage invites disaster. Second, as we've said before and will repeat again, programmers are optimists: tasks that look trivial to fix may sidetrack you for a long time from your primary task of refactoring, or worse, may get the code base into an unstable state from which you must backtrack in order to continue refactoring. The solution is simple: when you're refactoring or laying groundwork, focus obsessively on completing those steps *before* trying to enhance the code.

Fallacy: **It'll be faster to start from a clean slate than to fix this design.**

Putting aside the practical consideration that management will probably wisely forbid you from doing this anyway, there are many reasons why this belief is almost always wrong. First, if you haven't taken the time to understand a system, you are in no position to estimate how hard it will be to redesign, and probably will underestimate the effort vastly, given programmers' incurable optimism. Second, however ugly it may be, the current system *works*; a

main tenet of doing short Agile iterations is "always have working code," and by starting over you are immediately throwing that away. Third, if you use Agile methods in your redesign, you'll have to develop user stories and scenarios to drive the work, which means you'll need to prioritize them and write up quite a few of them to make sure you've captured at least the functionality of the current system. It would probably be faster to use the techniques in this chapter to write scenarios for just those parts of the system to be improved and drive new code from there, rather than doing a complete rewrite.

Does this mean you should *never* wipe the slate clean? No. As Rob Mee of Pivotal Labs points out, a time may come when the current codebase is such a poor reflection of the original design intent that it becomes a liability, and starting over may well be the best thing to do. (Sometimes this results from not refactoring in a timely way!) But in all but the most trivial systems, this should be regarded as the "nuclear option" when all other paths have been carefully considered and determined to be inferior ways to meet the customer's needs.

Pitfall: **Rigid adherence to metrics or "allergic" avoidance of code smells.**

In Chapter 8 we warned that correctness cannot be assured by relying on a single type of test (unit, functional, integration/acceptance) or by relying exclusively on quantitative code coverage as a measure of test thoroughness. Similarly, code quality cannot be assured by any single code metric or by avoiding any specific code smells. Hence the `metric_fu` gem inspects your code for multiple metrics and smells so you can identify "hot spots" where multiple problems with the same piece of code call for refactoring.

9.9 Concluding Remarks: Continuous Refactoring

A ship in port is safe, but that's not what ships are built for.

—Admiral Grace Murray Hopper

As we said in the opening of the chapter, modifying legacy code is not a task to be undertaken lightly, and the techniques required must be honed by experience. The first time is always the hardest. But fundamental skills such as refactoring help with both legacy code and new code, and as we saw, there is a deep connection among legacy code, refactoring, and testability and test coverage. We took code that was neither good nor testable—it scored poorly on complexity metrics and code smells, and isolating behaviors for unit testing was awkward—and refactored it into code that has much better metric scores, is easier to read and understand, and is easier to test. In short, we showed that *good methods are testable and testable methods are good.* We used refactoring to beautify existing code, but the same techniques can be used when performing the enhancements themselves. For example, if we need to add functionality to an existing method, rather than simply adding a bunch of lines of code and risk violating one or more SOFA guidelines, we can apply Extract Method to place the functionality in a new method that we call from the existing method. As you can see, this technique has the nice benefit that we already know how to develop new methods using TDD!

This observation explains why TDD leads naturally to good and testable code—it's hard for a method not to be testable if the test is written first—and illustrates the rationale behind the "refactor" step of Red–Green–Refactor. If you are refactoring constantly as you code, each individual change is likely to be small and minimally intrusive on your time and concentration, and your code will tend to be beautiful. When you extract smaller methods from

larger ones, you are identifying collaborators, describing the purpose of code by choosing good names, and inserting seams that help testability. When you rename a variable more descriptively, you are documenting design intent.

But if you continue to encrust your code with new functionality *without* refactoring as you go, when refactoring finally does become necessary (and it will), it will be more painful and require the kind of significant scaffolding described in Sections 9.2 and 9.3. In short, refactoring will suddenly change from a background activity that takes incremental extra time to a foreground activity that commands your focus and concentration at the expense of adding customer value.

Since programmers are optimists, we often think "That won't happen to me; I wrote this code, so I know it well enough that refactoring won't be so painful." But in fact, your code becomes legacy code the moment it's deployed and you move on to focusing on another part of the code. Unless you have a time-travel device and can talk to your former self, you might not be able to divine what you were thinking when you wrote the original code, so the code's clarity must speak for itself.

This Agile view of continuous refactoring should not surprise you: just as with development, testing, or requirements gathering, refactoring is not a one-time "phase" but an ongoing process. In Chapter 12 we will see that the view of continuous vs. phased also holds for deployment and operations.

It may be a surprise that the fundamental characteristics of Agile make it an excellent match to the needs of software maintenance. In fact, we can think of Agile as not having a development phase at all, but being in maintenance mode from the very start of its lifecycle!

Working with legacy code isn't exclusively about refactoring, but as we've seen, refactoring is a major part of the effort. The best way to get better at refactoring is to do it a lot. Initially, we recommend you browse through Fowler's refactoring book just to get an overview of the many refactorings that have been cataloged. We recommend the Ruby-specific version (Fields et al. 2009), since not all smells or refactorings that arise in statically-typed languages occur in Ruby; versions are available for other popular languages, including Java. We introduced only a few in this chapter; Figure 9.22 lists more. As you become more experienced, you'll recognize refactoring opportunities without consulting the catalog each time.

Code smells came out of the Agile movement. Again, we introduced only a few from a more extensive catalog; Figure 9.23 lists more. Good programmers don't deliberately create code with code smells; more often, the smells creep in as the code grows and evolves over time, sometimes beyond its original design. Pytel and Saleh's *Rails Antipatterns* (Pytel and Saleh 2010) and Tucker's treatment of code smells and refactoring in the context of contributing to open source software (Tucker et al. 2011) address these realistic situations.

We also introduced some simple software metrics; over four decades of software engineering, many others have been produced to capture code quality, and many analytical and empirical studies have been done on the costs and benefits of software maintenance. Robert Glass (Glass 2002) has produced a pithy collection of *Facts & Fallacies of Software Engineering*, informed by both experience and the scholarly literature and focusing in particular on the perceived vs. actual costs and benefits of maintenance activities.

Sandi Metz's *Practical Object-Oriented Design in Ruby* Metz 2012 covers object-oriented design from the perspective of minimizing the cost of change, and expands on many of the themes in this chapter with practical examples.

The other primary sources for this chapter are Feathers's excellent practical treatment of

Category	Refactorings		
Composing Methods	*Extract method*	Replace temp with method	Introduce explaining variable
	Replace method with method object	Inline temp	Split temp variable
	Remove parameter assignments	Substitute algorithm	
Organizing Data	self-encapsulate field	replace data value with object	change value to reference
	replace array/hash with Object	*Replace magic number with symbolic constant*	
Simplifying Conditionals	*Decompose Conditional*	Consolidate Conditional	Introduce Assertion
	Replace Conditional with Polymorphism	Replace Type Code with Polymorphism	Replace Nested Conditional with Guard Clauses
	Consolidate Duplicate Conditional Fragments	Remove Control Flag	Introduce Null Object
Simplifying Method Calls	Rename Method	Add Parameter	Separate Query from Modifier
	Replace Parameter with Explicit Methods	Preserve Whole Object	Replace Error Code with Exception

Figure 9.22: Several more of Fowler's refactorings, with the ones introduced in this chapter in italics.

Duplicated Code	Temporary Field	Large Class	Long Parameter List
Divergent Change	Feature Envy	Primitive Obsession	Metaprogramming Madness
Data Class	Lazy Class	Speculative Generality	Parallel Inheritance Hierarchies
Refused Bequest	Message Chains	Middle Man	Incomplete Library Class
Too Many Comments	Case Statements	Alternative Classes with Different Interfaces	

Figure 9.23: Several of Fowler's and Martin's code smells, with the ones introduced in this chapter in italics.

working with legacy code (Feathers 2004), Nierstrasz and Demeyer's book on reengineering object-oriented software (Nierstrasz et al. 2009), and of course, the Ruby edition of Fowler's classic catalog of refactorings (Fields et al. 2009).

Finally, John Ousterhout's *A Philosophy of Software Design* Ousterhout 2018 contains lots more practical advice for structuring software at the class and method level, with a view towards robustness and manageability. It's aimed at more advanced developers and is an excellent source of wisdom when you're ready to go beyond the introductory material in this chapter.

M. Feathers. *Working Effectively with Legacy Code*. Prentice Hall, 2004. ISBN 9780131177055.

J. Fields, S. Harvie, M. Fowler, and K. Beck. *Refactoring: Ruby Edition*. Addison-Wesley Professional, 2009. ISBN 0321603508.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999. ISBN 0201485672.

R. L. Glass. *Facts and Fallacies of Software Engineering*. Addison-Wesley Professional, 2002. ISBN 0321117425.

R. Green and H. Ledgard. Coding guidelines: Finding the art in the science. *Communications of the ACM*, 54(12):57–63, Dec 2011.

B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):466–471, 1978.

R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. ISBN 9780132350884.

S. Metz. *Practical Object-Oriented Design in Ruby: An Agile Primer (Addison-Wesley Professional Ruby)*. Addison-Wesley Professional, 2012. ISBN 0321721330. URL `http://poodr.com`.

O. Nierstrasz, S. Ducasse, and S. Demeyer. *Object-Oriented Reengineering Patterns*. Square Bracket Associates, 2009. ISBN 395233412X.

J. K. Ousterhout. *A Philosophy of Software Design*. Yaknyam Press, 2018.

C. Poole and J. W. Huisman. Using extreme programming in a maintenance environment. *Software, IEEE*, 18(6):42–50, 2001.

C. Pytel and T. Saleh. *Rails AntiPatterns: Best Practice Ruby on Rails Refactoring (Addison-Wesley Professional Ruby Series)*. Addison-Wesley Professional, 2010. ISBN 9780321604811.

A. Tucker, R. Morelli, and C. de Silva. *Software Development: An Open Source Approach (Chapman & Hall/CRC Innovations in Software Engineering and Software Development Series)*. CRC Press, 2011. ISBN 143981290X.

Notes

[1] `http://campfirenow.com`
[2] `http://basecamphq.com`
[3] `http://en.wikibooks.org/wiki/Ruby_Programming/RubyDoc`
[4] `http://api.rubyonrails.org`
[5] `http://paulschreiber.com/blog/2010/06/15/rake-task-extracting-database-contents/`
[6] `http://c2.com/doc/oopsla89/paper.html`
[7] `https://vimeo.com/24668095`
[8] `http://codeclimate.org`

10 Agile Teams

Frederick P. "Fred" Brooks, Jr. (1931–) wrote the classic software engineering book *The Mythical Man-Month* based on his years leading OS/360, the operating system for the highly successful IBM System/360 family of computers. This family was the first to feature instruction set compatibility across models, making it the first system to which the term "computer architecture" could be meaningfully applied. Brooks received the 1999 Turing Award for landmark contributions to computer architecture, operating systems, and software engineering.

There are no winners on a losing team, and no losers on a winning team.

—Fred Brooks, quoting North Carolina basketball coach Dean Smith, 1990

10.1 It Takes a Team: Two-Pizza and Scrum 302
10.2 Using Branches Effectively 304
10.3 Pull Requests and Code Reviews 309
10.4 Delivering the Backlog Using Continuous Integration 313
10.5 CHIPS: Agile Iterations 317
10.6 Reporting and Fixing Bugs: The Five R's 317
10.7 The Plan-And-Document Perspective on Managing Teams 319
10.8 Fallacies and Pitfalls 327
10.9 Concluding Remarks: From Solo Developer to Teams of Teams . . . 329

Prerequisites and Concepts

Prerequisites:

You should have a free GitHub account and be comfortable with the basic Git operations and commands for solo work: creating a new repo in your development environment and pushing it to GitHub, cloning an existing repo from GitHub to your development environment, adding and removing files in a repo, committing changes accompanied by descriptive log messages, and pushing your changes to GitHub. Even if your Integrated Development Environments (IDEs) provides buttons and menus for interacting with Git and GitHub, you need to be comfortable doing these operations from the command line.

To learn or refresh these skills, we recommend this basic Git guide[1] (developed for UC Berkeley CS61B Data Structures).

Concepts:

Whether Agile or Plan-and-Document, programming is now primarily a team sport. This chapter covers techniques that can help teams succeed in coordinating and delivering their work. All software teams rely on ***version control*** to manage code and configuration data, ***code reviews*** to ensure quality, ***release management*** to ship new versions, and ***change management*** while maintaining a shipped product, but the processes around these activities differ between Agile and P&D teams. The version of these concepts for Agile is:

- *"Two-pizza"* teams are four to nine people in size.
- Self-organizing teams follow the ***Scrum*** model, which relies on one teammate to act as the *Product Owner*, who represents the customer, and one to act as the *ScrumLead*, who acts as a buffer between the team and external distractions. These roles rotate between the team members over time.
- Code reviews occur continuously, often as the result of ***pull requests***, which start the process of integrating new team contributions into the mainline code.

For the Plan-and-Document lifecycle, you will become familiar with the same concepts from a different perspective:

- The ***project manager*** writes the contract, interfaces with the customer and upper management, recruits and manages the development team, resolves conflicts, and documents the plans for managing configurations and the project itself.
- While group sizes are similar to Agile, large teams can be created by combining groups into a hierarchy under the project manager, with each group having its own leader.

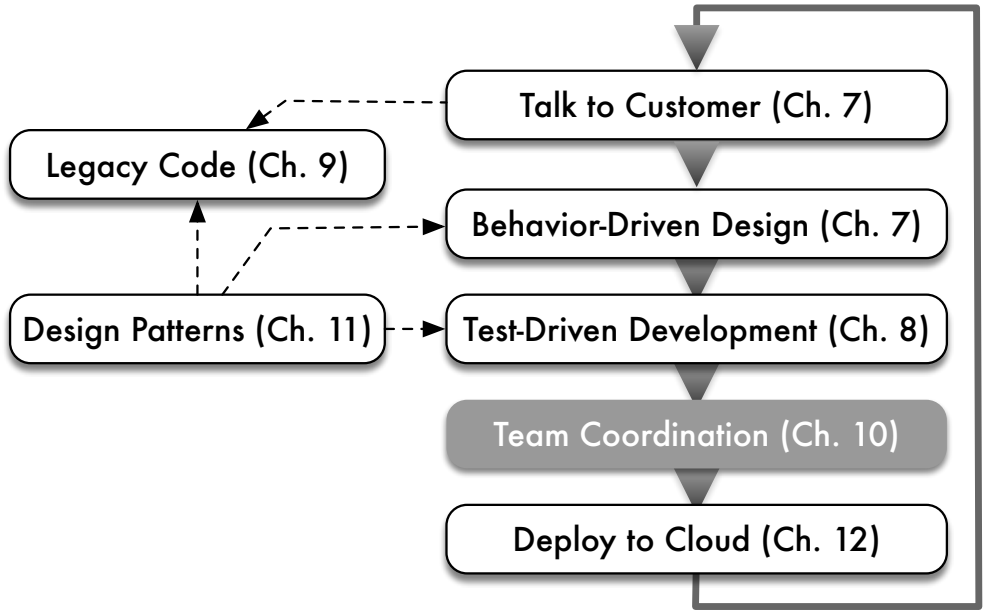

Figure 10.1: The Agile software lifecycle and its relationship to the chapters in this book. This chapter emphasizes evaluating the productivity of the team so as to come up with schedules that are more accurate by tracking velocity.

10.1 It Takes a Team: Two-Pizza and Scrum

> *The Six Phases of a Project: (1) Enthusiasm, (2) Disillusionment, (3) Panic, (4) Search for the guilty, (5) Punishment of the innocent, (6) Praise for non-participants.*
>
> —Dutch Holland (Holland 2004)

The days of the hero programmer are now past. Whereas once a brilliant individual could create breakthrough software, the rising bar on functionality and quality means software development now is primarily a team sport. Hence, success today means that not only do you have great design and coding skills, but that you work well with others and can help your team succeed. As the opening quote from Fred Brooks states, you cannot win if your team loses, and you cannot fail if your team wins.

Jeff Bezos, the CEO of Amazon who received his college degree in computer science, coined the two-pizza characterization of team size.

Hence, the first step of a software development project is to form and organize a team. As to its size, a "two-pizza" team—a group that can be fed by two pizzas in a meeting—is typical for SaaS projects. Our discussions with senior software engineers suggest the typical team size range varies by company, with four to nine developers being a typical range.

While there are many ways to organize a two-pizza software development, a popular one today is ***Scrum*** (Schwaber and Beedle 2001). Its frequent short meetings—15 minutes every day at the same place and time—inspire the name, when each team member answers three questions:

1. What have you done since yesterday?
2. What are you planning to do today?
3. Are there any impediments or stumbling blocks?

The benefit of these daily scrums is that by understanding what each team member is doing and has already completed, the team can identify work that would help others make faster progress. Indeed, daily scrums are sometimes referred to as *standups*, implying that the meeting should be kept short enough that everyone can remain standing the entire time.

When combined with the weekly or biweekly iteration model of Agile to collect the feedback from all the stakeholders, the Scrum organization makes it more likely that the rapid progress will be towards what the customers want. Rather than use the Agile term iteration, Scrum uses the term ***sprint***.

A scrum is held on every minor infraction in the sport of rugby. The game stops to bring the players together for a quick "meeting" in order to restart the game.

A Scrum has three main roles:

1. **Team**—A two-pizza size team that delivers the software.
2. **Scrum Lead**—A team member who acts as buffer between the Team and external distractions, keeps the team focused on the task at hand, enforces team rules, and removes impediments that prevent the team from making progress. One example is enforcing ***coding standards***, which are style guidelines that improve the consistency and readability of the code.
3. **Product Owner**—A team member (not the Scrum Lead) who represents the voice of the customer and prioritizes user stories.

Scrum relies on self-organization, and team members often rotate through different roles. For example, we recommend that each member rotate through the Product Owner role, changing on every iteration or sprint.

In any group working together, conflicts can occur around which technical direction the group should go. Depending in part on the personalities of the members of the team, they may not be able to quickly reach agreement. One approach to resolving conflicts is to start with a list on all the items on which the sides agree, as opposed to starting with the list of disagreements. This technique can make the sides see that perhaps they are closer together than they thought. Another approach is for each side to articulate the other's arguments. This technique makes sure both sides understand what the arguments are, even if they don't agree with some of them. This step can reduce confusion about terms or assumptions, which may be the real cause of the conflict.

Of course, such an approach requires great team dynamics. Everyone on the team should ideally feel ***psychological safety***—the belief that they will not be humiliated for voicing their ideas to the team, even if those ideas might turn out to be wrong. Indeed, a two-year study by Google[2] found that the most effective Google teams weren't the ones with the most senior engineers or the smartest people, but the teams with high psychological safety. One way Agile teams promote psychological safety is to do a short Retrospective meeting, often shortened to "retro," at the end of each iteration. A typical format for the retro focuses on Plus/Minus/Interesting (PMI): each team member writes down, perhaps anonymously at first, what they thought went well, went poorly, and was unusual or noteworthy (neither good nor bad) during the iteration. The PMI items are often not technical, for example, "When I brought up my concern about some new code to Armando, I felt he was dismissive about my idea" or "Dave really helped me with a bug I'd been chasing this week without making me feel stupid." All team members then review the items (and where appropriate reveal their identities, or say "I agree," or "I noticed that too"), noting especially if some items were raised by more than one team member. The PMI items can also be compared to the previous iteration's retro items, since one goal of Agile is continuous improvement.

Postfacto is a free Web-based tool from Pivotal Labs that facilitates retros.

The rest of this chapter focuses on coordinating the work of the team, and in particular the use of version control tools to support coordination. How is the code repository managed? What happens if team members accidentally make conflicting changes to a set of files? Which lines in a given file were changed when, and by whom were they changed? How does one developer work on a new feature without introducing problems into stable code? How does the team ensure the quality of the code and tests on an ongoing basis as more contibrutions accrete in the repository? As we will see, when software is developed by a team rather than an individual, version control can be used to address these questions using ***merging*** and ***branching***. Both tasks involve combining changes made by many developers into a single code base, a task that sometimes requires manual resolution of conflicting changes.

Summary: SaaS is a good match for two-pizza teams and Scrum, a self-organized small team that meets daily. Two team members take on the additional roles of Scrum Lead, who removes impediments and keeps the team focused, and Product Owner, who speaks for the customer. It can be helpful to follow structured strategies to resolve conflicts when they occur and to reflect on past work in a retrospective meeting.

■ ***Elaboration: Coding standards***

Coding standards or *stylesheets* are style guidelines that everyone on the team is expected to follow, usually related to indentation, variable naming, and so on. The goal is to improve the consistency and readability of the code. For example, here is one for Rails[3], and Google offers them for Python[4], JavaScript[5], and several other languages.

Self-Check 10.1.1. *True or False: Scrum is an appropriate methodology when it is difficult to plan ahead.*

◇ True: Scrum relies more on real-time feedback than on the traditional management approach of central planning with command and control. ■

10.2 Using Branches Effectively

Besides taking snapshots of your work and backing it up, version control also lets you manage multiple versions of a project's code base simultaneously, for example, to allow part of the team to work on an experimental new feature without disrupting working code, or to fix a bug in a previously-released version of the code that some customers are still using.

Branches are designed for such situations. Rather than thinking of commits as just a sequence of snapshots, we should instead think of a directed, acyclic *graph* of commits. When a new repo is created, by default it contains only a single branch, usually called the main branch, on which a linear sequence of commits is made. At any point in time, a new branch can be created that "splits off" from any commit of an existing branch, creating a copy of the code tree as it exists at that commit. As soon as a branch is created, that branch and the one from which it was split are separate: commits to one branch don't affect the other, though depending on project needs, commits in either may be merged back into the other. Indeed, branches can even be split off from other branches, but overly complex branching structures offer few benefits and are difficult to maintain. Finally, unlike a real tree branch, a repo branch can be merged back into another branch later—either the branch from which it split off, or some other branch. A branch can also be deleted, for example, if the project

Prior to June 2020, the main branch was usually called *master*; the usage is changing to avoid associations with the practice of slavery.

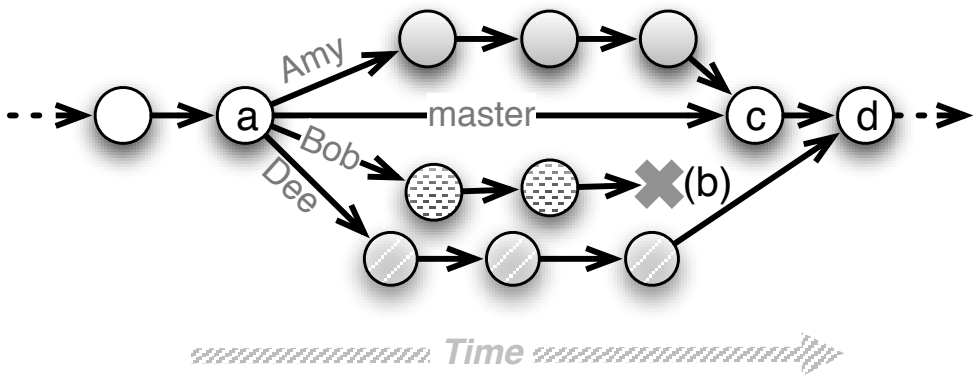

Figure 10.2: Each circle represents a commit. Amy, Bob and Dee each start branches based on the same commit (a) to work on different RottenPotatoes features. After several commits, Bob decides his feature won't work out, so he deletes his branch (b); meanwhile, Amy completes her work and merges her feature branch back into the main branch, creating the merge-commit (c). Finally, Dee completes her feature, but since the main branch has changed due to Amy's merge-commit (c), Dee has to do some manual conflict resolution to complete her merge-commit (d).

decides to abandon the work-in-progress that branch represents.

We highlight two common branch management strategies that can be used together or separately, both of which strive to ensure that the main branch always contains a stable working version of the code. Figure 10.2 shows a *feature branch*, which allows a developer or sub-team to make the changes necessary to implement a particular feature without affecting the main branch until the changes are complete and tested. If the feature is merged into the main branch and a decision is made later to remove it (perhaps it failed to deliver the expected customer value), the specific commits related to the merge of the feature branch can sometimes be undone, as long as there haven't been many changes to the main branch that depend on the new feature.

Topic branch is a generic term that may refer to feature, release, or bug fix branches.

Figure 10.3 shows how *release branches* are used to fix problems found in a specific release. They are widely used for delivering non-SaaS products such as libraries or gems whose releases are far enough apart that the main branch may diverge substantially from the most recent release branch. For example, the Linux kernel, for which developers check in thousands of lines of code per day, uses release branches to designate stable and long-lived releases of the kernel. Release branches often receive multiple merges from the development or main branch and contribute multiple merges to it. Release branches are less common in delivering SaaS because of the trend toward continuous integration/continuous deployment (Section 1.4): if you deploy several times per week, the deployed version won't have time to get out of sync with the main branch, so you might as well deploy directly from the main branch. We discuss continuous deployment further in Chapter 12.

Figure 10.4 shows some commands for manipulating Git branches. At any given time, the *current branch* is whichever one you're working on in your copy of the repo. Since in general each copy of the repo contains all the branches, you can quickly switch back and forth between branches in the same repo (but see Fallacies and Pitfalls for an important caveat about doing so).

Small teams working on a common set of features commonly use a *shared-repository* model for managing the repo: one particular copy of the repo, referred to as the *origin* repo, is designated as authoritative, and all developers agree to push their changes to the origin

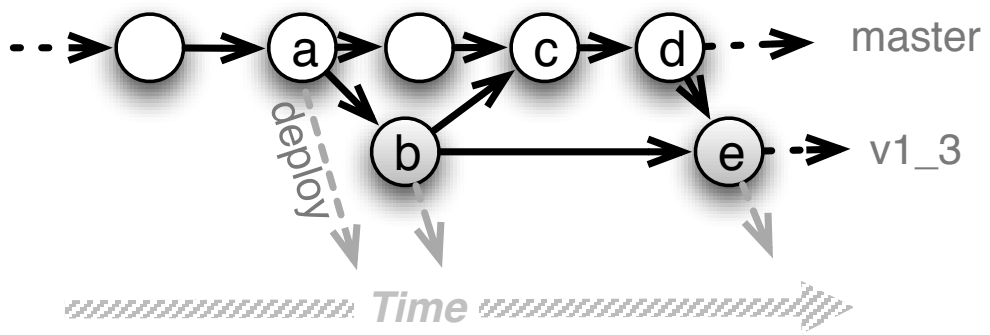

Figure 10.3: (a) A new release branch is created to "snapshot" version 1.3 of RottenPotatoes. A bug is found in the release and the fix is committed in the release branch (b); the app is redeployed from the release branch. The commit(s) containing the fix are merged into the main branch (c), but the code in the main branch has evolved sufficiently from the code in the release that manual adjustments to the bug fix need to be made. Meanwhile, the dev team working on the main branch finds a critical security flaw and fixes it with one commit (d). The specific commit containing the security fix can be merged into the release branch (e) using `git cherry-pick`, since we don't want to apply any other main branch changes to the release branch except for this fix.

Many earlier VCSs such as Subversion supported *only* the shared-repository model of development, and the "one true repo" was often called the *master*, a term that means something quite different in Git.

and periodically pull from the origin to get others' changes. Famously, Git itself doesn't care which copy of the repo is authoritative—any developer can *pull* changes from or *push* changes to any other developer's copy of that repo if the repo's permissions are configured to allow it—but for small teams, it's convenient (and conventional) for the origin repo to reside in the cloud on a service such as GitHub. Each team member can *clone* the origin repo onto their development computer, do their work, make their commits on their local clone, and periodically push their commits to the origin. From the point of view of each developer's local clone, the origin is one of possibly several *remote* copies of the repo, or simply "remotes." In the shared-repository model, the origin repo is usually the only remote. Section 10.4 describes scenarios in which there may be multiple remotes.

PaaS (Platform-as-a-Service) deployment servers such as Heroku often appear as a Git remote; pushing code to that remote deploys the pushed version of the app.

As Figure 10.4 shows, the `git push` and `git pull` commands usually specify which copy of the repository and which branch should be involved in a push or pull operation. If Amy commits changes to her clone of the repo, those changes aren't visible to her teammate Bob until she does a push and Bob does a pull.

This raises the possibility of a *merge conflict* scenario. Returning to Figure 10.2, suppose that Dee's feature branch results in changes to some of the same files as Amy's feature branch. At the commit marked (c), Amy has successfully merged her changes into the main branch. When Dee tries to push her changes (d), she will initially be prevented from doing so because additional commits have occurred in the origin repo since she last pulled (probably at point (a) when creating her branch). Dee must bring her copy of the repo up-to-date with respect to the origin before she can push her changes. One way to do this is for her to switch back to her main branch, then run `git pull origin main` to get the latest commits on main from the origin repo; her copy of the repo now looks the same as it looked to Amy right after point (c). Now Dee can try to merge her branch changes back into main. If Dee's branch changes different files than Amy's branch, or if they change different parts of the same file that are far apart, the merge will succeed and Dee can then push her merged main branch back to the shared repo (`git push origin main`).

`git pull` actually combines two separate commands: `git fetch`, which copies new commits from the origin, and `git merge`, which tries to reconcile the commits with those in the branch on the local clone.

But if Dee and Amy had edited parts of the same file that were within a few lines of each other, as in Figure 10.5, Git will conclude that there is no safe way to automatically

- `git branch`
  List existing branches in repo, indicating current branch with `*`. If you're using `sh` or a `bash`-derived shell on a Unix-like system, placing the following in `~/.profile` will make the shell prompt display the current Git branch when you're in a Git repo directory:
  https://gist.github.com/17f75e5697e5bca7a9ce2be75d012a65

```
export PS1="[`git branch --no-color 2>/dev/null | \
  sed -e '/^[^*]/d' -e 's/* \(.*\)/\1/'`]% "
```

- `git checkout` *name*
  Switch to existing branch *name*.
- `git branch` *name*
  If branch *name* exists, switch to it; otherwise create a new branch called *name* without switching to it. The shortcut `git checkout -b` *name [commit-id]* creates and switches to a new branch based on *commit-id*, which defaults to most recent commit in the current branch.
- `git push` *[repo] [branch]*
  Push the changes (commits) on *branch* to remote repository *repo*. (The first time you do this for a given branch, it creates that branch on the remote *repo*.) With no arguments, pushes the current local branch to the current branch's remote, or the remote called `origin` by default.
- `git pull` *[repo] [branch]*
  Fetches and merges commits from branch *branch* on the remote *repo* into your local repo's **current** branch *(even if the current branch's name doesn't match the branch name you're pulling from—beware!)*. To fetch a remote branch `foo` for which you have no corresponding local branch, first use `git checkout -b foo` to create a local branch by that name and switch to it, then `git pull origin foo`. With no arguments, *repo* and *branch* default to the values of `git config branch.`*currentbranch*`.remote` and `git config branch.`*currentbranch*`.merge` respectively, which are automatically set up by certain Git commands and can be changed with `git branch --track`. If you setup a new repo in the usual way, *repo* defaults to `origin` and *branch* defaults to `main`.
- `git remote show` *[repo]*
  If *repo* omitted, show a list of existing remote repos. If *repo* is the name of an existing remote repo, shows branches located at *repo* and which of your local branches are set up to track them. Also shows which local branches are not up-to-date with respect to *repo*.
- `git merge` *branch*
  Merge all changes from *branch* into the current branch.
- `git rebase` *source-branch*
  Try to rewrite history as if the current branch had originated from the latest commit on *source-branch*. You can also specify a specific commit-ID instead of the name of a source branch. Useful in pull requests since it shifts the work of conflict resolution from the target branch's maintainer to the feature branch's maintainer.
- `git cherry-pick` *commits*
  Rather than merging all changes (commits) from a given branch, apply *only* the changes introduced by each of the named *commits* to the current branch.
- `git checkout` *branch file1 file2...*
  For each file, merge the differences in *branch*'s version of that file into the current branch's version of that file.

Figure 10.4: Common Git commands for handling branches and merging. Branch management involves merging; Figure 10.6 tells how to undo merges gone awry.

https://gist.github.com/708b2be6cbc71f1f3d499e683a4474fb

```
1 Roses are red,
2 Violets are blue.
3 <<<<<<< HEAD:poem.txt
4 I love GitHub,
5 =======
6 ProjectLocker rocks,
7 >>>>>>> 77976da35a11db4580b80ae27e8d65caf5208086:poem.txt
8 and so do you.
```

Figure 10.5: When Bob tries to merge Amy's changes, Git inserts conflict markers in `poem.txt` to show a merge conflict. Line 3 marks the beginning of the conflicted region with <<<; everything until === (line 5) shows the contents of the file in `HEAD` (the latest commit in Bob's local repo) and everything thereafter until the end-of-conflict marker >>>(line 7) shows Amy's changes (the file as it appears in Amy's conflicting commit, whose commit-ID is on line 7). Lines 1,2 and 8 were either unaffected or merged automatically by Git.

create a version of the file that reflects both sets of changes, and it will leave a conflicted *and uncommitted* version of the file with conflict markers (<<< and >>>) in Dee's main branch. Dee must now manually edit that file and add and commit the manually edited version to complete the merge, after which she can push the merged main back to the shared repo. In the next section, we will discuss an alternative process for preventing merge conflicts before they occur. If a merge goes badly awry, Figure 10.6 provides some mechanisms for partially or fully undoing the results of the merge. Figure 10.7 lists some useful Git commands to help keep track of who did what and when. Figure 10.8 shows some convenient notational alternatives to the cumbersome 40-digit Git commit-IDs.

Finally, don't overlook the importance of a *scratch branch*, which is a branch that is never intended to be merged back into the mainline code. You can create a scratch branch to explore code changes such as exploring a spike (Section 7.4) or dry-running a radical change such as upgrading to a major new version of your app framework.

Whichever branches you create, if those branches are pushed to the main repo, over time the number of branches will grow. GitHub has a user interface for viewing and pruning stale (inactive) branches, which helps keep your codebase manageable.

Summary of branching:

- Small teams typically use a "shared-repo" model, in which pushes and pulls use a single authoritative copy of the repo. In Git, the authoritative copy is often referred to as the `origin` repo and is stored in the cloud on GitHub or on an internal company server.
- Branches allow variation in a code base. For example, feature branches support the development of new features without destabilizing working code, and release branches allow fixing bugs in previous releases whose code has diverged from the main line of development.
- Merging changes from one branch into another (for example, from a feature branch back into the main branch) may result in conflicts that must be manually resolved.
- With Agile and SaaS, feature branches are usually short-lived and release branches are uncommon.

Self-Check 10.2.1. *Describe a scenario in which merges could go in both directions—changes in a feature branch merged into the main branch, and changes in the main branch merged into a feature branch. (In Git, this is called a* crisscross merge*.)*

◇ Diana starts a new branch to work on a feature. Before she completes the feature, an important bug is fixed and the fix is merged into the main branch. Because the bug is in a part of the code that interacts with Diana's feature, she merges the fix from main into her own feature branch. Finally, when she finishes the feature, her feature branch is merged back into main. ■

10.3 Pull Requests and Code Reviews

> *There is a really interesting group of people in the United States and around the world who do social coding now. The most interesting stuff is not what they do on Twitter, it's what they do on GitHub.*
>
> —Al Gore, former US Vice President, 2013

Section 10.7 describes the use of design reviews or code reviews to improve quality of the software product. You may be surprised to learn that most companies using Agile methods do not perform formal design or code reviews. But perhaps more surprising is that experienced Agile companies' code is *better and more frequently* reviewed compared to companies that do formal code reviews.

For the explanation of this paradox, recall the basic idea of extreme programming: every good programming practice is taken to an extreme. Section 2.2 already described one form of "extreme code review" in the form of pair programming, in which the navigator continuously reviews the code being entered by the driver. In this section we describe another form of code review, in which the rest of the team has frequent opportunities to review the work of their colleagues. Our description follows the process used at GitHub, where formal code reviews are rare.

When a developer (or a pair) has finished work on a branch, rather than directly merging the branch into the main, the developer makes a ***pull request***, or PR for short, asking that the branch's changes be merged into (usually) the main branch. All developers sharing the repo see each PR, and each has the responsibility to determine how merging those changes might affect their own code. If anyone has a concern, an online discussion coalesces around the PR. For example, GitHub's user interface allows any developer to make comments either on the PR overall or on specific lines of particular files. This discussion might result in changes to the code in question before the PR is accepted and merged; any further changes made on the PR's topic branch and pushed to GitHub will automatically be reflected in the PR, so the PR process is really a form of code review. And since many PRs typically occur each day, these "mini-reviews" are occurring continuously, so there is no need for special meetings.

Merge request is an alternative name for pull request.

The PR therefore serves as a way to focus a code review discussion on a particular set of changes. That said, a PR shouldn't be opened until the developer is confident their code is ready. A few examples of such preparation include:

- The code to be merged should be well covered by tests, all of which should be passing.
- Documentation (design documents, the `README` file, the project wiki, and so on) has been updated if necessary to explain new design decisions or changes to important configuration files (such as the `Gemfile` for Ruby projects).

- Any temporary or non-essential files that were versioned during development of the code have been removed from version control.
- Steps have been taken to eliminate or minimize merge conflicts that will occur when the PR is accepted and merged.

The last item above can be tricky, because as the previous section explained, it's not uncommon for a merge to encounter conflicts that must be manually resolved. Indeed, when you open a PR, GitHub checks and informs you whether the merge would require manual conflict resolution. Such a scenario motivates the possible use of *rebasing*, an operation in which you tell Git to make the world look as if you had branched from a later commit. For example, if there have been 3 new commits on main since the time you branched off of it, and you now rebase your branch on top of main, Git will *first* apply those 3 new commits to the original state of your branch, and *then* try to apply your own commits. This latter step may cause conflicts if the 3 commits on the main branch touched some of the same files your changes have touched. If so, Git generally requires you to resolve each conflict as it is detected, before proceeding with the rebase, and allows you to abort the rebase entirely if things get too ugly. But once you have resolved those conflicts, merging your branch back to main is guaranteed not to cause any new conflicts (unless, of course, additional commits to main happened while you were rebasing). In other words, from the point of view of trying to merge changes into a shared main branch, rebasing before merging forces you to resolve the conflicts at rebasing time, thus saving work for whoever will merge your branch back into the main branch.

There is an important caveat to rebasing. By its very nature, rebasing rewrites history, by making the world look as if your branch had been created from a different commit than it actually was, and by rewriting the commit history of the branch itself. This rewriting of history can occur in one of three scenarios:

Commit squashing is an optional rebasing step in which the branch's commits are "squashed" into a single commit that can be easily backed out to undo the effects of merging the PR.

1. You have not yet pushed to the shared repo. The history of your branch exists only in your copy of the repo, so rewriting that history does not affect the team.
2. You have pushed to the shared repo, but no one else has made additional changes based on your branch. This is the common case when using branch-per-feature, since there is normally no reason for one developer to base work on another developer's commits in a feature branch. You may need to use the `--force` flag to `git push` when pushing the branch, to indicate your acknowledgment that you're changing the shared repo's view of history.
3. You have pushed to the shared repo, and others have based work off of your changes. Anyone who has based their work off of your branch commits will now be out of sync since their history doesn't match the shared repo's history.

Case 3 is rare, but if it occurs, you should coordinate carefully with your team *before* forcing a push to avoid others' repos getting out of sync with the shared copy. One good practice is to construct feature branch names in some way that signals that others should not build off of those commits, for example by prepending the developer's initials to the branch name or using a standard naming convention such as `feature-`*xxx* for feature branch names.

An alternative to rebasing is merging: running `git pull origin main` in your feature branch at any time will update your clone from the origin repo, then merge any new changes

- `git reset --hard ORIG_HEAD`
  Revert your repo to last committed state just before the merge.
- `git reset --hard HEAD`
  Revert your repo to last committed state.
- `git checkout` *commit* `--` *[file]*
  Restore a file, or if omitted the whole repo, to its state at *commit* (see Figure 10.8 for ways to refer to a commit besides its 40-digit SHA-1 hash). Can be used to recover files that were previously deleted using `git rm`.
- `git revert` *commit*
  Reverts the changes introduced by *commit*. If that commit was the result of a merge, effectively undoes the merge, and leaves the current branch in the state it was in before the merge. Git tries to back out just the changes introduced by that commit without disturbing other changes since that commit, but if the commit happened a long time ago, manual conflict resolution may be required.

Figure 10.6: When a merge goes awry, these commands can help you recover by undoing all or part of the merge.

`git blame` *[file]*	Annotate each line of a file to show who changed it last and when.
`git diff` *[file]*	Show differences between current working version of *file* and last committed version.
`git diff` *branch [file]*	Show differences between current version of *file* and the way it appears in the most recent commit on *branch* (see Section 10.2).
`git log` *[ref..ref] [files]*	Show log entries affecting all *files* between the two commits specified by the *ref*s (which must be separated by exactly two dots), or if omitted, entire log history affecting those files.
`git log --since="`*date*`"` *files*	Show the log entries affecting all *files* since the given date (examples: `"25-Dec-2019"`, `"2 weeks ago"`).

Figure 10.7: Git commands to help track who changed what file and when. Many commands accept the option `--oneline` to produce a compact representation of their reports. If an optional *[file]* argument is omitted, default is "all tracked files." Note that all these commands have *many* more options, which you can see with `git help` *command*.

from the main branch into your feature branch. Compared with rebasing, this approach is nondestructive because it doesn't rewrite history, but it also adds a lot of extra commits (the merge commits) to your feature branch, which can make it tricky to reconstruct the history of the feature branch using `git log`. Atlassian has an excellent set of tutorials[6] covering this and many other Git-related topics.

All this having been said, if you're breaking down your user stories into tasks of manageable size (Section 7.4) and doing frequent deployments (Section 12.4), messy merges and rebases should rarely be necessary in Agile development.

`HEAD`	The most recently committed version on the current branch.
`HEAD~`	The prior commit on the current branch (`HEAD~`*n* refers to the *n*'th previous commit).
`ORIG_HEAD`	When a merge is performed, `HEAD` is updated to the newly-merged version, and `ORIG_HEAD` refers to the commit state before the merge. Useful if you want to use `git diff` to see how each file changed as a result of the merge.
`1dfb2c~2`	2 commits prior to the commit whose ID has `1dfb2c` as a unique prefix.
"*branch*`@{`*date*`}`"	The last commit prior to *date* (see Figure 10.7 for date format) on *branch*, where `HEAD` refers to the current branch.

Figure 10.8: Convenient ways to refer to certain commits in Git commands, rather than using a full 40-digit commit ID or a unique prefix of one. `git rev-parse` ***expr*** **resolves any of the above expressions into a full commit ID.**

Summary of merge management for small teams:

1. By opening a pull request to merge a feature branch rather than performing the merge directly, the rest of the team is notified of the proposed changes and has a chance to do an on-the-spot code review around the pull request before it is accepted.
2. Rebasing rewrites history by "rewinding" a branch to originate from a different commit than it actually does, and then trying to apply the branch's commits, thereby forcing the branch maintainer to resolve conflicts that *would* result if the un-rebased branch were merged. A common use for rebasing is to allow subsequent creation of a pull request that is guaranteed to be free of merge conflicts.
3. Because it rewrites history after the fact, rebasing should only be used when you can be sure that no one else has based their additional work on that branch's commits.

■ ***Elaboration: When to open a pull request***

Our advice above has been to open a PR when you're confident that the code is ready for review, but some teams instead open a PR much earlier as a "draft PR," as the code is in progress. The requester knows the PR won't be merged, but this way the rest of the team can comment on the code as it evolves, rather than waiting until it's fully ready. The PR can remain open as the code evolves in response to comments. Once the PR is judged ready for final review prior to merge, the "draft" designation is removed. The "early draft PR" approach is consistent with the XP philosophy: if code reviews are good, do them as early as possible and on fine-grained evolution of the code. The disadvantage is that it may create additional work for other team members compared to the simpler "Please look over my code now" approach, especially if the developer opening the PR is experienced and unlikely to benefit from very-early-stage code review.

Self-Check 10.3.1. *True or false: If you attempt* `git push` *and it fails with a message such as "Non-fast-forward (error): failed to push some refs," this means some file contains a merge conflict between your repo's version and the origin repo's version.*

◇ Not necessarily. It just means that your copy of the repo is missing some commits that are present in the origin copy, and until you merge in those missing commits, you won't be allowed to push your own commits. Merging in these missing commits *may* lead to a merge conflict, but frequently does not. ■

10.4 Delivering the Backlog Using Continuous Integration

As Section 7.4 explained, the ***backlog*** is the somewhat pessimistic term for the work remaining to be done during the current Agile iteration. That section described how to prioritize and estimate the difficulty of the iteration's planned work, and briefly introduced Pivotal Tracker as a way to track the work. While there is no single "correct" workflow for Agile teams, in this section we describe a widely-used workflow and suggest best practices for using it effectively. We assume that the stories have already been prioritized and assigned points during an Iteration Planning Meeting, as Section 7.4 described.

The key idea behind delivering the backlog is ***continuous integration*** (CI), which minimizes the time between when changes are made on a feature branch and when those changes are merged into the main branch and deployed for customer review. A good CI workflow for Agile 2-pizza teams starts with a shared team repo and the use of pull requests to integrate changes from feature branches into the main branch. We introduce two new concepts here that are central to CI. The first is that of a service such as Travis[7], whose job is to run your complete test suite, usually in the cloud, each time significant changes are made during development of a new feature. The rationale is that while you're continuously testing the code for the feature you're developing, you may not be taking the time to run the full test suite, which can take minutes or even tens of minutes for large projects. Somewhat confusingly, such services are usually called CI services, even though technically CI refers not just to running the test suite but to the entire workflow by which changes are integrated into mainline code. Most CI services can be connected directly to a GitHub repository and automatically run CI every time code is pushed to any branch in that repository.

A full CI run may include compilation (for compiled languages) and running operational tests (Chapter 12) beyond the scope of the specific feature being developed.

The second concept is that of a staging server, which is configured as similarly as possible to the production server but usually much smaller in scale. The purpose of a staging server is to provide a safe place to deploy new features for customer review before they are deployed in production. The staging server may not even be a specific persistent server like the production server, but an ephemeral one "spun up" just to give the customer the opportunity to see a specific feature in action. A staging server has its own copy of the database containing test data (possibly extracted from real customer data), and is usually off-limits to outside users.

In this workflow, we distinguish three copies of a repo, each of whose main branches is represented by a thick horizontal line in Figure 10.9. Initially, we will describe the workflow from the point of view of the *origin* repo, which is owned by some team member and shared in the cloud by the team, and some developer's local *clone* of the origin. (We will use "developer" to mean either an individual team member or a pair working on a story.) The local repo is created by running `git clone` with the URL of the origin repo. From the point of view of the local repo, the origin repo is one of possibly several *remotes*, and usually the default remote when there is more than one. The following numbered steps are keyed to the numbers in the figure, and annotated to indicate the associated state of the story in Pivotal Tracker story.

1. On the main branch of local, the developer uses `git pull origin main` to ensure she has the most up-to-date version of main.

2. The developer creates a new feature branch for the story (Section 10.2). At this point the story state changes from Unstarted to Started.

3. The developer writes tests and code for the feature (Chapters 7 and 8), committing frequently. Periodically pushing the feature branch's commits to the origin repo (`git`

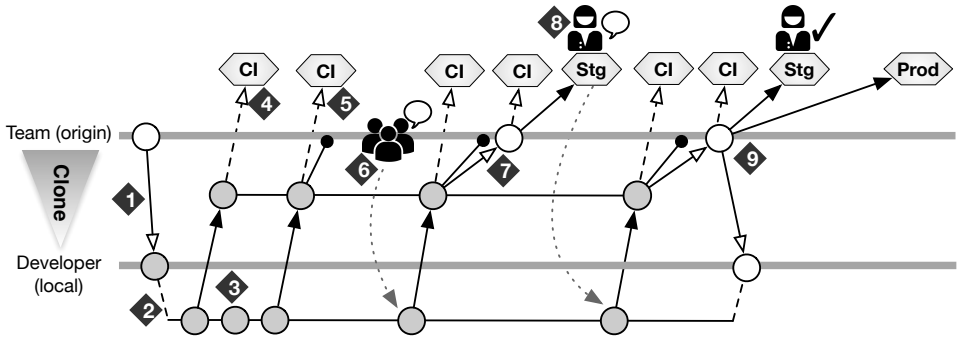

Figure 10.9: A basic workflow for a coordinated team delivering features when the team itself "owns" the app being developed. The boxes along the top indicate suggested interpretation of the Pivotal Tracker story states (started, finished, delivered, accepted) relative to events in the workflow.

`push origin` *feature-branch*) keeps a copy of the feature branch on the origin repo up-to-date with the one on the local feature branch.

4. This team has their workflow configured so that any push to the origin repo will automatically trigger an external CI run on whatever branch was pushed.

5. When the code and tests are ready, the developer marks the story Finished, and opens a pull request. Note that the PR relates the topic branch *on the origin repo* to the main branch *also on the origin repo*, that is, the developer must have pushed the most recent commits on her local topic branch to its counterpart on origin.

6. Other team members review and comment on the PR (Section 10.3), in this case leading to required changes by the developer, who makes the changes and reopens the PR (or opens a new one).

7. The revised PR is accepted and the changes are merged into the origin repo's main branch, triggering another CI run, since the main branch now includes not only this PR but possibly other developers' PRs that have been merged since Step 1.

8. Assuming CI passes, the origin's main branch is deployed to a staging server and the story is marked Delivered. The customer can now review and comment on the new feature. If the customer requests revisions, another round of changes, PR, and merging follows.

With the above workflow, in a team with several developers (or pairs) many stories and feature branches may be "in flight" at the same time.

What happens next depends on which repo is the *base repo* for the app, that is, the definitive repo from which the production app is released and which is stewarded by the app's owners. If the development team in question owns the app, then the team's origin repo may be the app's base repo as well. But if the app is owned by other developers, their repo is the base repo, and this team's origin is just used for development. In that case, a pull request can be opened from the origin's main branch to the base repo's main to merge the changes, and goes through the usual process of review.

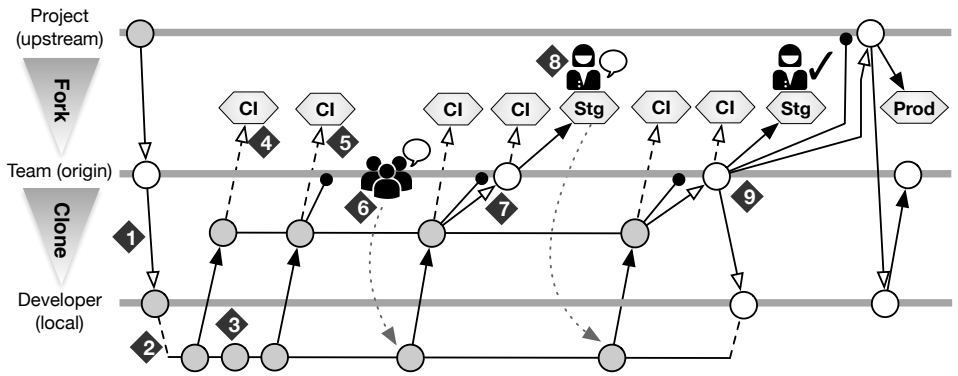

Figure 10.10: Variation of Figure 10.9 when the team does not own the app being developed. The final steps now involve coordinating with the upstream repos' stewards to get the changes merged there, and then update the team's origin repo (via a two-step process, as the text describes) to get the latest upstream changes.

Figure 10.10 shows this "fork-and-pull" collaboration model with the upstream or base repo shown as the top line. The subteam's origin repo, sometimes also called the *head repo*, is a fork of the base repo and opens PRs to merge head repo changes into the base repo. Just as feature branch developers may periodically rebase (Section 10.3) against their origin repo's main branch to get the latest changes and avoid later merge conflicts, the head repo may pull changes from the base repo for the same reason, but because of the way Git works, these changes cannot propagate directly from the base repo on GitHub to its fork. Instead, some developer must pull the changes into their own local copy of the repo, then push the changes to the appropriate branch of the origin repo. The official GitHub tutorial on forking[8] gives detailed steps for keeping a fork up-to-date with an upstream repo, though in your authors' opinion this alternative tutorial[9] is both clearer and more concise.

Fork historically meant a schism in which one team makes a copy of another team's source code and starts developing it independently, often against the wishes of the original authors. GitHub overloaded the term to mean "creating your own copy of a repo to which you want to contribute but lack push access."

These workflows should make even more clear why Section 7.4 recommends keeping stories simple: A 1-point story is one whose implementation strategy is mostly known to the team, so it can be delivered quickly and predictably, and if mistakes are made, they can be easily undone with little time being wasted. As with all aspects of Agile, a short cycle with quick feedback is better than spending a lot of time making large changes that carry more uncertainty and risk. A small story also means a simple and short-lived branch, speeding up pull requests and often eliminating the need for rebasing.

Still, no matter how careful your team is, sometimes a pull request may need to be revised before it's accepted, or the customer may partially reject a feature leading to the branch being modified and the PR being reopened, as both figures show. Such scenarios are part and parcel of Agile development, but to keep your repo clean and your team sane, we recommend mitigating such "loops" by following these best practices for delivering the backlog:

1. *Follow your own advice.* The goal of the Iteration Planning Meeting is to determine points and priorities for this iteration. Those decisions should be respected during the iteration, but if they need to be revisited, that should be a team effort rather than a unilateral decision by one developer.

2. *Work on one story at a time.* One developer or pair should finish a story before starting stories that require other changes, unless they run into impediments that prevent further progress on delivering that story.

3. *A sustainable pace is more important than total points.* Rather than delivering 5 points all at once at the iteration's end, deliver 1 point per day for 5 days, giving the rest of the team (and the customer) the opportunity to review your contributions as they come in.

Summary of pull-request-based workflow using branch-per-feature:

1. Pick a story to implement and mark it "Started"
2. Create and switch to a new feature branch for the story in your local copy of the repo
3. Develop code and tests using BDD and TDD on the branch, committing frequently
4. When the branch is ready for review and all tests are passing, open a pull request
5. Based on team feedback on the pull request, continue making changes on the branch until all feedback has been addressed
6. Merge the branch into the main or main branch
7. Mark the story "Finished"
8. The story will be marked "Delivered" when it is ready for the customer to try out, and "Accepted" when the customer signs off

Summary of Delivering the Backlog via CI:

- Continuous integration is about frequently integrating new code and tests into the mainline. Each integration provides an opportunity to get feedback from the team via "mini code reviews" during pull requests, and from the customer via frequent deployment to a staging server.
- Central to CI is the continuous running of tests as code is developed, usually using a separate service such as Travis.
- When a subteam is working on features, they may fork the upstream repo and do their teamwork on their development repo, issuing a final pull request to the upstream repo when their own workflow is complete.
- CI relies heavily on automation. For example, workflows can be constructed that automatically trigger a testing run optionally followed by automatic deployment to a staging server when commits are pushed to a specific repo or branch.

■ ***Elaboration: Further Automating CI***

You can configure your workflow so that CI not only runs automatically on every push, but if the push passes CI, it is automatically deployed to an ephemeral staging server for customer review. This way, both code reviews and customer reviews can occur before the formal merge of the PR integrates these changes into the main branch. Some companies such as Salesforce go even further: if any test fails in CI, the CI tool performs binary searches across recent commits to pinpoint which specific commit introduced a new bug, and automatically opens and assigns a bug report for the developer responsible for that commit (Hansma 2011).

Self-Check 10.4.1. *Can you think of a scenario in which it makes sense for the team to review a PR for a particular story, but it* does not *make sense to deploy the story to staging for the customer's feedback?*

◇ Often, a customer-requested feature is broken down into many separate stories, some of which do not result in new functionality visible to the customer, such as adding a new model without yet creating any views for it. The customer's feedback will be needed as soon as there is a way to interact with the feature, even if incomplete, but not before. ■

10.5 CHIPS: Agile Iterations

■ ***CHIPS 10.5: Agile Iterations***

`https://github.com/saasbook/hw-agile-iterations`

Perform one or two complete iterations of the Agile lifecycle by planning and tracking features for a SaaS app, adding code and tests for the features, deploying the features to staging and production, and using tools to track test coverage and code quality throughout.

10.6 Reporting and Fixing Bugs: The Five R's

Inevitably, bugs happen. If you're lucky, they are found before the software is in production, but production bugs happen too. Everyone on the team must agree on processes for managing the phases of the bug's lifecycle:

1. **R**eporting a bug
2. **R**eproducing the problem, or else **R**eclassifying it as "not a bug" or "won't be fixed"
3. Creating a **R**egression test that demonstrates the bug
4. **R**epairing the bug
5. **R**eleasing the repaired code

Any stakeholder may find and **r**eport a bug in server-side or client-side SaaS code. A member of the development or QA team must then **r**eproduce the bug, documenting the environment and steps necessary to trigger it. This process may result in **r**eclassifying the bug as "not a bug" for various reasons:

- This is not a bug but a request to make an enhancement or change a behavior that is working as designed
- This bug is in a part of the code that is being undeployed or is otherwise no longer supported
- This bug occurs only with an unsupported user environment, such as a very old browser lacking necessary features for this SaaS app
- This bug is already fixed in the latest version (uncommon in SaaS, whose users are always using the latest version)

Large projects for widely-used software may use considerably more complex bug tracking systems, such as the open-source Bugzilla[10].

"Severity 1" bugs at Amazon.com require the responsible engineers to initiate a conference call within 15 minutes of learning of the bug—a stricter responsiveness requirement than for on-call physicians! (Bodík et al. 2006)

Once the bug is confirmed as genuine and reproducible, it's entered into a bug management system. A plethora of such systems exists, but the needs of many small to medium teams can be met by a tool you're already using: Pivotal Tracker allows marking a story as a Bug rather than a Feature, which assigns the story zero points but otherwise allows it to be tracked to completion just like a regular user story. An advantage of this tool is that Tracker manages the bug's lifecycle for you, so existing processes for delivering user stories can be readily adapted to fixing bugs. For example, fixing the bug must be prioritized relative to other work; in a waterfall process, this may mean prioritization relative to other outstanding bugs while in the maintenance phase, but in an Agile process it usually means prioritization relative to developing new features from user stories. Using Tracker, the Product Manager can move the bug story above or below other stories based on the bug's severity and impact on the customer. For example, bugs that may cause data loss in production will get prioritized very high.

The next step is **r**epair, which always begins with *first* creating the *simplest possible* automated test that fails in the presence of the bug, and *then* changing the code to make the test(s) pass green. This should sound familiar to you by now as a TDD practitioner, but this practice is true even in non-TDD environments: *no bug can be closed out without a test.* Depending on the bug, unit tests, functional tests, integration tests, or a combination of these may be required. *Simplest* means that the tests depend on as few preconditions as possible, tightly circumscribing the bug. For example, simplifying an RSpec unit test would mean minimizing the setup preceding the test action or in the `before` block, and simplifying a Cucumber scenario would mean minimizing the number of `Given` or `Background` steps. These tests usually get added to the regular regression suite to ensure the bug doesn't recur undetected. A complex bug may require multiple commits to fix; a common policy in BDD+TDD projects is that commits with failing or missing tests shouldn't be merged to the main development branch until the tests pass green.

Many bug tracking systems can automatically cross-reference bug reports with the commit-IDs that contain the associated fixes and regression tests. For example, using GitHub's service hooks[11], a commit can be annotated with the story ID of the corresponding bug or feature in Tracker, and when that commit is pushed to GitHub, the story is automatically marked as Delivered. Depending on team protocol and the bug management system in use, the bug may be "closed out" either immediately by noting which release will contain the fix or after the release actually occurs.

As we will see in Chapter 12, in most Agile teams releases are very frequent, shortening the bug lifecycle.

Summary: the 5 R's of bug fixing

- A bug must be **r**eported, **r**eproduced, demonstrated in a **r**egression test, and **r**epaired, all before the bug fix can be **r**eleased.
- No bug can be closed out without an automated test demonstrating that we really understand the bug's cause.
- Bugs that are really enhancement requests or occur only in obsolete versions of the code or in unsupported environments may be reclassified to indicate they're not going to be fixed.

Self-Check 10.6.1. *Why do you think "bug fix" stories are worth zero points in Tracker even though they follow the same lifecycle as regular user stories?*
◇ A team's velocity would be artificially inflated by fixing bugs, since they'd get points for implementing the feature in the first place and then more points for actually getting the implementation right. ■

Self-Check 10.6.2. *True or false: a bug that is triggered by interacting with another service (for example, authentication via Twitter) cannot be captured in a regression test because the necessary conditions would require us to control Twitter's behavior.*
◇ False: integration-level mocking and stubbing, for example using the FakeWeb gem[12] or the techniques described in Section 8.4, can almost always be used to mimic the external conditions necessary to reproduce the bug in an automated test. ■

Self-Check 10.6.3. *True or false: a bug in which the browser renders the wrong content or layout due to JavaScript problems might be reproducible manually by a human being, but it cannot be captured in an automated regression test.*
◇ False: tools such as Jasmine and Webdriver (Section 6.8) can be used to develop such tests. ■

10.7 The Plan-And-Document Perspective on Managing Teams

In Plan-And-Document processes, project management starts with the project manager. Project managers are the bosses of the projects:

- They write the contract proposal to win the project from the customer.
- They recruit the development team from existing employees and new hires.
- They typically write team members' performance reviews, which shape salary increases.
- From a Scrum perspective (Section 10.1), they act as Product Owner—the primary customer contact—*and* they act as Scrum Lead, as they are the interface to upper management and they procure resources for the team.
- As we saw in Section 7.9, project managers also estimate costs, make and maintain the schedule, and decide which risks to address and how to overcome or avoid them.
- As you would expect for Plan-And-Document processes, project managers must document their project management plan. Figure 10.11 gives an outline of Project Management Plans from the corresponding IEEE standard.

As a result of all the these responsibilities, project managers receive much of the blame if projects have problems. Quoting a textbook author from his introduction to project management:

> *However, if a post mortem were to be conducted for every [problematic] project, it is very likely that a consistent theme would be encountered: project management was weak.*
>
> — Pressman 2010

We cover four major tasks for project managers to increase their chances of being successful:

1. Project overview
  1.1 Project summary
    1.1.1 Purpose, scope and objectives
    1.1.2 Assumptions and constraints
    1.1.3 Project deliverables
    1.1.4 Schedule and budget summary
  1.2 Evolution of the plan
2. References
3. Definitions
4. Project context
  4.1 Process model
  4.2 Process improvement plan
  4.3 Infrastructure plan
  4.4 Methods, tools and techniques
  4.5 Product acceptance plan
  4.6 Project organization
    4.6.1 External interfaces
    4.6.2 Internal interfaces
    4.6.3 Authorities and responsibilities
5. Project planning
  5.1 Project initiation
    5.1.1 Estimation plan
    5.1.2 Staffing plan
    5.1.3 Resource acquisition plan
    5.1.4 Project staff training plan
  5.2 Project work plans
    5.2.1 Work activities
    5.2.2 Schedule allocation
    5.2.3 Resource allocation
    5.2.4 Budget allocation
    5.2.5 Procurement plan
6. Project assessment and control
  6.1 Requirements management plan
  6.2 Scope change control plan
  6.3 Schedule control plan
  6.4 Budget control plan
  6.5 Quality assurance plan
  6.6 Subcontractor management plan
  6.7 Project closeout plan
7. Product delivery
8. Supporting process plans
  8.1 Project supervision and work environment
  8.2 Decision management
  8.3 Risk management
  8.4 Configuration management
  8.5 Information management
    8.5.1 Documentation
    8.5.2 Communication and publicity
  8.6 Quality assurance
  8.7 Measurement
  8.8 Reviews and audits
  8.9 Verification and validation

Figure 10.11: Format of a project management plan from the IEEE 16326-2009 ISO/IEC/IEEE Systems and Software Engineering–Life Cycle Processes–Project Management standard.

1. Team size, roles, space, communication
2. Managing people and conflicts
3. Inspections and metrics
4. Configuration management

1. Team size, roles, space, and communication. The Plan-and-Document processes can scale to larger sizes, where group leaders report to the project manager. However, each subgroup typically stays the size of the two-pizza teams we saw in Section 10.1. Size recommendations are three to seven people (Braude and Berstein 2011) to no more than ten (Sommerville 2010). Fred Brooks gave us the reason in Chapter 7: adding people to the team increases parallelism, but also increases the amount of time each person must spend communicating. These team sizes are reasonable considering the fraction of time spent communicating.

Given we know the size of the team, members of a subgroup in Plan-and-Document processes can be given different roles in which they are expected to lead. For example (Pressman 2010):

- Configuration management leader
- Quality assurance leader
- Requirements management leader
- Design leader
- Implementation leader

One surprising result is that the type of space for the team to work in affects project management. One study found that collocating the team in open space could double productivity (Teasley et al. 2000). The reasons include that team members had easy access to each other for both coordination of their work and for learning, and they could post their work artifacts on the walls so that all could see. Another study of teams in open space concludes:

> *One of the main drivers of success was the fact that the team members were at hand, ready to have a spontaneous meeting, advise on a problem, teach/learn something new, etc. We know from earlier work that the gains from being at hand drops off significantly when people are first out of sight, and then most severely when they are more than 30 meters apart.*
>
> — Allen and Henn 2006

While the team relies on email and texting for communicating and shares information in wikis and the like, there is also typically a weekly meeting to help coordinate the project. Recall that the goal is to minimize the time spent communicating unnecessarily, so it is important that the meetings be effective. Below is our digest of advice from the many guidelines found on the Web on how to have efficient meetings. We use the acronym SAMOSAS as a memory device; surely bringing a plate of them will make for an effective meeting!

Samosas are a popular stuffed deep-fried snack from India.

- **S**tart and stop meeting on time.
- **A**genda created in advance of meeting; if there is no agenda, then cancel the meeting.

- **M**inutes must be recorded so everyone can recall results afterwards; the first agenda item is finding a note taker.
- **O**ne speaker at a time; no interruptions when another is speaking.
- **S**end material in advance, since people read much faster than speakers talk.
- **A**ction items at end of meeting, so people know what they should do as a result of the meeting.
- **S**et the date and time of the next meeting.

2. Managing people and conflicts. Thousands of books have been written on how to manage people, but the two most useful ones that we have found are *The One Minute Manager* and *How to Win Friends and Influence People* (Blanchard and Johnson 1982; Carnegie 1998). What we like about the first book is that it offers short quick advice. Be clear about the goals of what you want done and how well it should be done, but leave it up to the team member how to do it to encourage creativity. When meeting with individuals to review progress, start with positive feedback to help build their confidence. Then, be honest with them about what is not going well, and what they need to do to fix it. Finally, conclude with positive feedback and encouragement to continue improving their work. What we like about the second book is that it helps teach the art of persuasion, to get people to do what you think should be done without ordering them to do it. These skills also help persuade people you *cannot* command: your customers and your management.

Both books are helpful when it comes to resolving conflicts within a team. Conflicts are not necessarily bad, in that it can be better to have the conflict than to let the project crash and burn. Intel Corporation labels this attitude *constructive confrontation.* If you have a strong opinion that a person is proposing the wrong thing technically, you are obligated to bring it up, even to your bosses. The Intel culture is to speak up even if you disagree with the highest ranked people in the room.

If conflict continues, given that Plan-and-Document processes have a project manager, that person can make the final decision. One reason the US made it to the moon in the 1960s is that a leader of NASA, Wernher von Braun, had a knack for quickly resolving conflicts on close decisions. His view was that picking an option arbitrarily but quickly was frequently better, since the choice was roughly 50-50, so that the project could move ahead rather than take the time to carefully collect all the evidence to see which choice was slightly better.

However, once a decision is made, the teams needs to embrace it and move ahead. The Intel motto for this resolution is *disagree and commit*: "I disagree, but I am going to help even if I don't agree."

3. Inspections and metrics. Inspections like ***design reviews*** and ***code reviews*** allow feedback on the system even before everything is working. The idea is that once you have a design and initial implementation plan, you are ready for feedback from developers beyond your team. Design and code reviews follow the Waterfall lifecycle in that each phase is completed in sequence before going on to the next phase, or at least for the phases of a single iteration in Spiral or RUP development.

A design review is a meeting in which the authors of program present its design. The goal of the review is to improve software quality by benefiting from the experience of the people attending the meeting. A code review is held once the design has been implemented.

This peer-oriented feedback also helps with knowledge exchange within the organization and offers coaching that can help the careers of the presenters.

Shalloway suggests that formal design code reviews are often too late in the process to make a big impact on the result (Shalloway 2002). He recommends to instead have earlier, smaller meetings that he calls "approach reviews." The idea is to have a few senior developers assist the team in coming up with an approach to solve the problem. The group brainstorms about different approaches to help find a good one.

If you plan to do a formal design review, Shalloway suggests that you first hold a "mini-design review" after the approach has been selected and the design is nearing completion. It involves the same people as before, but the purpose is to prepare for the formal review.

The formal review itself should start with a high-level description of what the customers want. Then give the architecture of the software, showing the APIs of the components. It will be important to highlight the design patterns used at different levels of abstraction (see Chapter 11). You should expect to explain *why* you made the decisions, and whether you considered plausible alternatives. Depending on the amount of time and the interests of those at the meeting, the final phase would be to go through the code of the implemented methods. At all these phases, you can get more value from the review if you have a concrete list of questions or issues that you would like to hear about.

One advantage of code reviews is that they encourage people outside your team to look at your comments as well as your code. As we don't have a tool that can enforce the advice from Chapter 9 about making sure the comments raise the level of abstraction, the only enforcing mechanism is the code review.

In addition to reviewing the code and the comments, inspections can give feedback on every part of the project in Plan-and-Document processes: the project plan, schedule, requirements, testing plan, and so on. This feedback helps with ***verification and validation*** of the whole project, to ensure that it is on a good course. There is even an IEEE standard on how to document the verification and validation plan for the project, which Figure 10.12 shows.

Like the algorithmic models for cost estimation (see Section 7.9), some researchers have advocated that software metrics could replace inspections or reviews to assess project quality and progress. The idea is to collect metrics across many projects in organization over time, establish a baseline for new projects, and then see how the project is doing compared to baseline. This quote captures the argument for metrics:

> *Without metrics, it is difficult to know how a project is executing and the quality level of the software.*
>
> — Braude and Berstein 2011

Below are sample metrics that can be automatically collected:

- Code size, measured in thousands of lines of code (*KLOC*) or in function points (Section 7.9).
- Effort, measured in person-months spent on project.
- Project milestones planned versus fulfilled.
- Number of test cases completed.
- Defect discovery rate, measured in defects discovered via testing per month.

1. Purpose
2. Referenced documents
3. Definitions
4. V&V overview
   - 4.1 Organization
   - 4.2 Top-level schedule
   - 4.3 Integrity level scheme
   - 4.4 Resources summary
   - 4.5 Responsibilities
   - 4.6 Tools, techniques, and methods
5. V&V processes
   - 5.1 Common V&V Processes, Activities and Tasks
   - 5.2 System V&V Processes, Activities and Tasks
     - 5.2.1 Acquisition Support
     - 5.2.2 Supply Planning
     - 5.2.3 Project Planning
     - 5.2.4 Configuration Management
     - 5.2.5 Stakeholder Requirements Definition
     - 5.2.6 Requirements Analysis
     - 5.2.7 Architectural Design
     - 5.2.8 Implementation
     - 5.2.9 Integration
     - 5.2.10 Transition
     - 5.2.11 Operation
     - 5.2.12 Maintenance
     - 5.2.13 Disposal
   - 5.3 Software V&V Processes, Activities and Tasks
     - 5.3.1 Software Concept
     - 5.3.2 Software Requirements
     - 5.3.3 Software Design
     - 5.3.4 Software Construction
     - 5.3.5 Software Integration Test
     - 5.3.6 Software Qualification Test
     - 5.3.7 Software Acceptance Test
     - 5.3.8 Software Installation and Checkout (Transition)
     - 5.3.9 Software Operation
     - 5.3.10 Software Maintenance
     - 5.3.11 Software Disposal
   - 5.4 Hardware V&V Processes, Activities and Tasks
     - 5.4.1 Hardware Concept
     - 5.4.2 Hardware Requirements
     - 5.4.3 Hardware Design
     - 5.4.4 Hardware Fabrication
     - 5.4.5 Hardware Integration Test
     - 5.4.6 Hardware Qualification Test
     - 5.4.7 Hardware Acceptance Test
     - 5.4.8 Hardware Transition
     - 5.4.9 Hardware Operation
     - 5.4.10 Hardware Maintenance
     - 5.4.11 Hardware Disposal
6. V&V reporting requirements
   - 6.1 Task reports
   - 6.2 Anomaly reports
   - 6.3 V&V final report
   - 6.4 Special studies reports (optional)
   - 6.5 Other reports (optional)
7. V&V administrative requirements
   - 7.1 Anomaly resolution and reporting
   - 7.2 Task iteration policy
   - 7.3 Deviation policy
   - 7.4 Control procedures
   - 7.5 Standards, practices, and conventions
8. V&V test documentation requirements

Figure 10.12: Outline of a plan for System and Software Verification and Validation from the IEEE 1012-2012 Standard.

- Defect repair rate, measured in defects fixed per month.

Other metrics can be derived from these so as to normalize the numbers to help compare results from different projects: KLOC per person-month, defects per KLOC, and so on.

The problem with this approach is that there is little evidence of correlation between these metrics that we can automatically collect and project outcomes. Ideally, the metrics would correlate and we could have much finer-grained understanding than comes from the occasional and time consuming inspections. This quote captures the argument de-emphasizing metrics:

> *However, we are still quite a long way from this ideal situation, and there are no signs that automated quality assessment will become a reality in the foreseeable future*
>
> — Sommerville 2010

4. Configuration management. Configuration management includes four varieties of changes, three of which we have seen before. The first is ***version control***, sometimes also called *source and configuration management* (SCM), described in Sections 10.2–10.4. This variety keeps track of versions of components as they are changed. The second, *system building*, is closely related to the first. Tools like `make` assemble the compatible versions of components into an executable program for the target system. The third variety is ***release management,*** which we cover in Chapter 12. The last is ***change management***, which comes from change requests made by customers and other stakeholders to fix bugs or to improve functionality (see Section 9.7).

As you surely expect by now, IEEE has a standard for Configuration Management. Figure 10.13 shows its table of contents.

Table of Contents

1. Overview
  1.1 Scope
  1.2 Purpose
2. Definitions, acronyms, and abbreviations
  2.1 Definitions
  2.2 Acronyms and abbreviations
3. Tailoring
4. Audience
5. The configuration management process
6. CM planning lower-level process
  6.1 Purpose
  6.2 Activities and tasks
7. CM management lower-level process
  7.1 Purpose
  7.2 Activities and tasks
8. Configuration identification lower-level process
  8.1 Purpose
  8.2 Activities and tasks
9. Configuration change control lower-level process
  9.1 Purpose
  9.2 Activities and Tasks
10. Configuration status accounting lower-level process
  10.1 Purpose
  10.2 Activities and tasks
11. CM configuration auditing lower-level process
  11.1 Purpose
  11.2 Activities and Tasks
12. Interface control lower-level process
  12.1 Purpose
  12.2 Activities and Tasks
13. Supplier configuration item control lower-level process
  13.1 Purpose
  13.2 Activities and Tasks
14. Release management lower-level process
  14.1 Purpose
  14.2 Activities and tasks

Figure 10.13: A table of contents for the IEEE 828-2012 Standard for Configuration Management in Systems and Software Engineering.

Summary: In Plan-and-Document processes:

- Project managers are in charge: they write the contract, recruit the team, and interface with the customer and upper management.
- The project manager documents the project plan and configuration plan, along with the verification and validation plan that ensures that other plans are followed!
- To limit time spent communicating, groups are three to ten people. They can be composed into hierarchies to form larger teams reporting to the project manager, with each group having its own leader.
- Guidelines for managing people include giving them clear goals but empowering them, and starting with the positive feedback in reviews but being honest about shortcomings and how to overcome them.
- While conflicts need to be resolved, they can be helpful in finding the best path forward for a project.
- Inspections like design reviews and code reviews let outsiders give feedback on the current design and future plans. Such reviews allow the to team benefit from the experience of others. They are also a good way to check if good practices are being followed and if the plans and documents are sensible.
- Configuration management is a broad category that includes change management while maintaining a product, version control of software components, system building of a coherent working program from those components, and release management to ship the product to customers.

Self-Check 10.7.1. *Compare the size of teams in Plan-and-Document processes versus Agile processes.*
◇ Plan-and-Document processes can form hierarchies of subgroups to create a much larger project, but each subgroup is basically the same size as a "two-pizza" team for Agile. ■

Self-Check 10.7.2. *True or False: Design reviews are meetings intended to improve the quality of the software product using the wisdom of the attendees, but they also result in technical information exchange and can be highly educational for junior members of the organization, whether presenters or just attendees.*
◇ True. ■

10.8 Fallacies and Pitfalls

***Fallacy:* If a software project is falling behind schedule, you can catch up by adding more people to the project.**

The main theme of Fred Brooks's classic book, *The Mythical Man-Month*, is that not only does adding people not help, it makes it worse. The reason is twofold: it takes a while for new people to learn about the project, and as the size of the project grows, the amount of

communication increases, which can reduce the time available for people to get their work done. His summary, which some call Brooks's Law, is

> *Adding manpower to a late software project makes it later.*
>
> —Fred Brooks, Jr.

Pitfall: **Dividing work based on the software stack rather than on features.**

It's less common than it used to be to divide the team into a front-end specialist, back-end specialist, customer liaison, and so forth, but it still happens. Your authors and others believe that better results come from having each team member deliver *all* aspects of a chosen feature or story—Cucumber scenarios, RSpec tests, views, controller actions, model logic, and so on. Especially when combined with pair programming, having each developer maintain a "full stack" view of the product spreads architectural knowledge around the team.

Fallacy: **It's fine to make simple changes on the main branch.**

Programmers are optimists. When we set out to change our code, we always think it will be a one-line change. Then it turns into a five-line change; then we realize the change affects another file, which has to be changed as well; then it turns out we need to add or change existing tests that relied on the old code; and so on. For this reason, *always* create a feature branch when starting new work. Branching with Git is nearly instantaneous, and if the change truly does turn out to be small, you can delete the branch after merging to avoid having it clutter your branch namespace.

Pitfall: **Forgetting to add files to the repo.**

If you create a new file but forget to add it to the repo, *your* copy of the code will still work but when others pull your changes your code won't work for them. Use `git status` regularly to see the list of Untracked Files, and use the `.gitignore`[13] file to avoid being warned about files you never want to track, such as binary files or temporary files.

Pitfall: **Versioning files that shouldn't be versioned.**

If a file isn't required to run the code, it probably shouldn't be in the repo: temporary files, binary files, log files, and so on should not be versioned. If files of test data are versioned, they should be part of a proper test suite. Files containing sensitive information such as API keys should *never* be checked into GitHub in plaintext (i.e. without encryption). If the files must be checked in, they should be encrypted at rest.

Pitfall: **Accidentally stomping on changes after merging or switching branches.**

If you do a pull or a merge, or if you switch to a different branch, some files may suddenly have different contents on disk. If any such files are already loaded into your editor, the versions being edited will be *out of date*, and even worse, if you now save those files, you will either overwrite merged changes or save a file that isn't in the branch you think it is. The solution is simple: *before* you pull, merge or switch branches, make sure you commit all current changes; *after* you pull, merge or switch branches, reload any files in your editor that may be affected—or to be really safe, just quit your editor before you commit. Be careful

too about the potentially destructive behavior of certain Git commands such as `git reset`, as described in "Gitster" Scott Chacon's informative and detailed blog post[14].

Pitfall: **Letting your copy of the repo get too far out of sync with the origin (authoritative) copy.**

It's best not to let your copy of the repo diverge too far from the origin, or merges (Section 10.2) will be painful. You should update frequently from the origin repo before starting work, and if necessary, rebase incrementally so you don't drift too far away from the main branch.

Fallacy: **Since each subteam is working on its own branch, we don't need to communicate regularly or merge frequently.**

Branches are a great way for different team members to work on different features simultaneously, but without frequent merges and clear communication of who's working on what, you risk an increased likelihood of merge conflicts and accidental loss of work when one developer "resolves" a merge conflict by deleting another developer's changes.

Pitfall: **Making commits too large.**

Git makes it quick and easy to do a commit, so you should do them frequently and make each one small, so that if some commit introduces a problem, you don't have to also undo all the other changes. For example, if you modified two files to work on feature A and three other files to work on feature B, do two separate commits in case one set of changes needs to be undone later. In fact, advanced Git users use `git add` with specific files, rather than `git add .` which adds every file in the current directory, to "cherry pick" a subset of changed files to include in a commit. And don't forget that no one else will see the commit until you use `git push` to propagate them to the team's origin repo.

10.9 Concluding Remarks: From Solo Developer to Teams of Teams

> *The first 90% of the code accounts for the first 10% of the development time. The remaining 10% of the code accounts for the other 90% of the development time.*
>
> —Tom Cargill, quoted in *Programming Pearls*, 1985

The history of version control systems mirrors the movement towards distributed collaboration among "teams of teams," with two-pizza teams emerging as a popular unit of cohesiveness. From about 1970–1985, the original Unix ***Source Code Control System*** (SCCS) and its longer-lived descendant ***Revision Control System*** (RCS) required the repo and all development to stay on the same computer (which might be a multi-user system) and disallowed simultaneous editing of the same file by different developers. The ***Concurrent Versions System*** (CVS) and ***Subversion*** introduced simultaneous editing and branches, but only a single repo. Git completed the decentralization by allowing any copy of a repo to push or pull from any other, enabling completely decentralized "teams of teams," and by making branching and merging much quicker and easier than its predecessors. Today, distributed collaboration is the norm: rather than a large distributed team, fork-and-pull allows a large number of Agile two-pizza teams to make independent progress, and the use of Git to support such efforts has become ubiquitous. The two-pizza team size makes it easier for a team to stay organized than the giant programming teams possible in Plan-and-Document. The

decentralized approach also distributes responsibility for project planning and cost esimation more than P&D, which relies on the project manager to make the time and cost estimates, assess risks, and to run the project so that it delivers the product on time and on budget with the required functionality.

We have previously seen two examples of processes in which the P&D and Agile versions comprise the same skills and steps, but sequenced differently. Test-driven development uses the same elements as conventional code writing followed by debugging, but in a different order. Agile iterations include the same elements as a waterfall project, but in a different order. Team coordination is a third example: Agile teams use the same processes as P&D teams—releases, code reviews, customer reviews, cost and effort estimation, assignment of different parts of the coding task to different developers—but in a different order. Agile proponents believe the techniques in this chapter can help an agile team avoid many of the pitfalls that have made software projects infamous for being late and over budget. Checking in continuously with other developers (via PRs) and customers (via frequent deployments to staging) during each iteration guides your team into spending its resources most effectively and is more likely to result in software that makes customers happy within the time and cost budget. A disciplined workflow using version control allows developers to make progress on many fronts simultaneously without interfering with each others' work, and also allows disciplined and systematic management of the bug lifecycle.

Finally, as with any experience, you should reflect on what went well, what didn't go well, and what you would do differently. It is not a sin to make a mistake, as long as you learn from it; the sin is making the same mistake repeatedly. Many Agile teams' end-of-iteration Retrospective meeting allows this learning to happen incrementally each week and makes the team more cohesive over time, rather than waiting until the end of a long project to determine what could have gone better.

For more comprehensive details on this chapter's topics, we recommend these resources:

- You can find very detailed descriptions of Git's powerful features in *Version Control With Git* (Loeliger 2009), which takes a more tutorial approach, and in the free Git Community Book[15], which is also useful as a thorough reference on Git. For detailed help on a specific command, use `git help` *command*, for example, `git help branch`; but be aware that these explanations are for reference, not tutorial.
- Atlassian has an excellent set of tutorials[16] covering many Git-related topics, including rebasing.
- Many medium-sized projects that don't use Pivotal Tracker, or whose bug-management needs go somewhat beyond what Tracker provides, rely on the Issues feature built into every GitHub repo. The Issues system allows each team to create appropriate "labels" for different bug types and priorities and create their own "bug lifecycle" process.

T. J. Allen and G. Henn. *The Organization and Architecture of Innovation: Managing the Flow of Technology*. Butterworth–Heinemann, 2006.

K. H. Blanchard and S. Johnson. *The One Minute Manager*. William Morrow, Cambridge, MA, 1982.

P. Bodík, A. Fox, M. I. Jordan, D. Patterson, A. Banerjee, R. Jagannathan, T. Su, S. Tenginakai, B. Turner, and J. Ingalls. Advanced tools for operators at Amazon.com. In *First Workshop on Hot Topics in Autonomic Computing (HotAC'06)*, Dublin, Ireland, June 2006.

E. Braude and M. Berstein. *Software Engineering: Modern Approaches, Second Edition*. John Wiley and Sons, 2011. ISBN 9780471692089.

D. Carnegie. *How to Win Friends and Influence People*. Pocket, 1998.

S. Hansma. Go fast and don't break things: Ensuring quality in the cloud. In *Workshop on High Performance Transaction Systems (HPTS 2011)*, Asilomar, CA, Oct 2011. Summarized in Conference Reports column of USENIX ;login 37(1), February 2012.

D. Holland. *Red Zone Management*. WinHope Press, 2004. ISBN 0967140188.

J. Loeliger. *Version Control with Git: Powerful Tools and Techniques for Collaborative Software Development*. O'Reilly Media, 2009. ISBN 0596520123.

R. Pressman. *Software Engineering: A Practitioner's Approach, Seventh Edition*. McGraw-Hill, 2010. ISBN 0073375977.

K. Schwaber and M. Beedle. *Agile Software Development with Scrum (Series in Agile Software Development)*. Prentice Hall, 2001. ISBN 0130676349.

A. Shalloway. *Agile Design and Code Reviews*. 2002. URL `http://www.netobjectives.com/download/designreviews.pdf`.

I. Sommerville. *Software Engineering, Ninth Edition*. Addison-Wesley, 2010. ISBN 0137035152.

S. Teasley, L. Covi, M. S.Krishnan, and J. S. Olson. How does radical collocation help a team succeed? In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 339–346, Philadelphia, Pennsylvania, December 2000.

Notes

[1] `https://sp19.datastructur.es/materials/guides/using-git.html`
[2] `https://rework.withgoogle.com/guides/understanding-team-effectiveness/steps/introduction/`
[3] `https://github.com/bbatsov/rails-style-guide`
[4] `http://google-styleguide.googlecode.com/svn/trunk/pyguide.html`
[5] `http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml`
[6] `https://www.atlassian.com/git/tutorials`
[7] `http://travis-ci.org`
[8] `https://help.github.com/en/github/getting-started-with-github/fork-a-repo`
[9] `https://gist.github.com/Chaser324/ce0505fbed06b947d962`
[10] `http://mozilla.org`
[11] `http://github.com/`
[12] `http://fakeweb.rubyforge.org`
[13] `http://book.git-scm.com/4_ignoring_files.html`
[14] `http://progit.org/2011/07/11/reset.html`
[15] `http://book.git-scm.com`
[16] `https://www.atlassian.com/git/tutorials`