Things fall apart

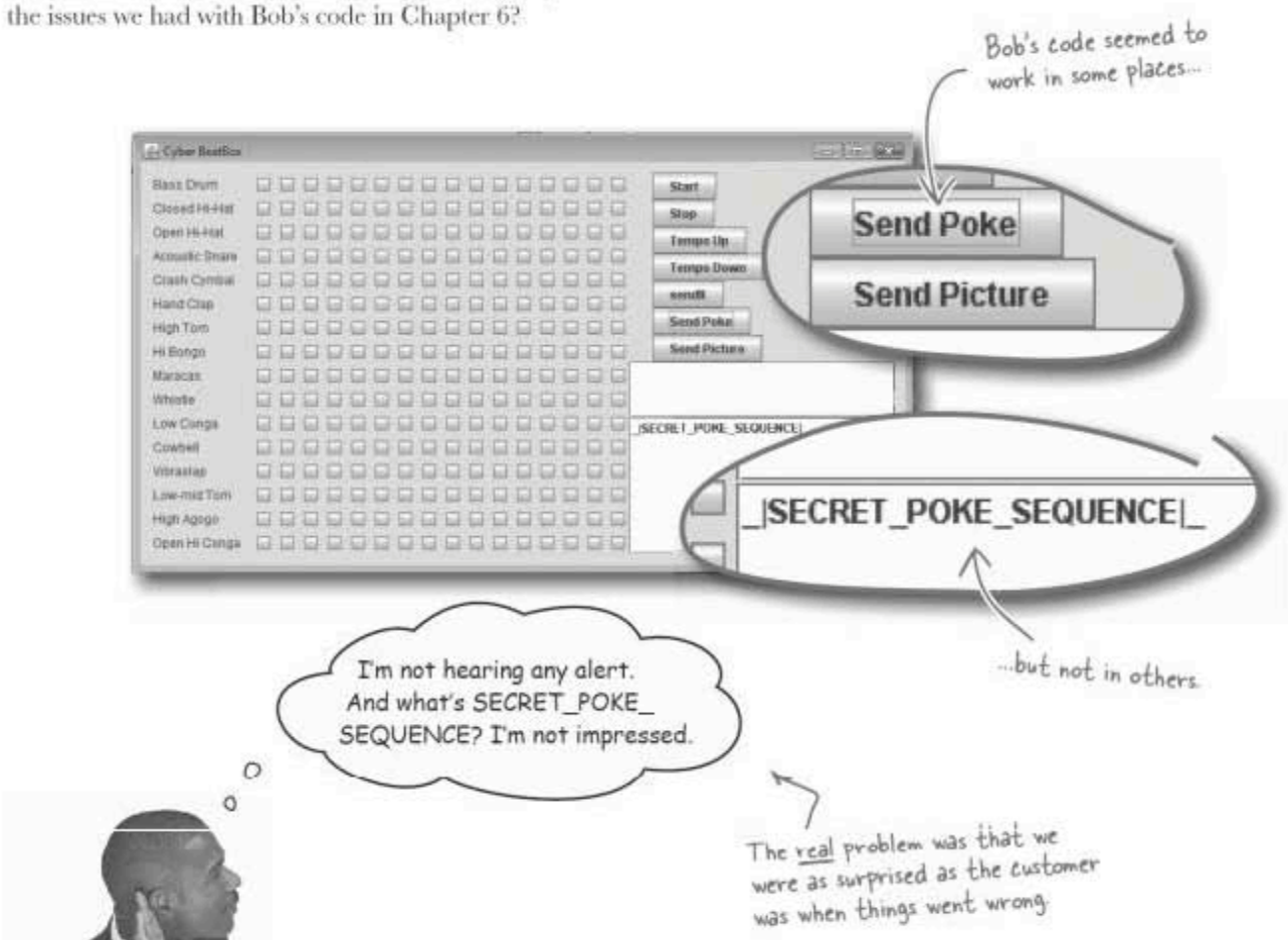
Sometimes even the best developer breaks the build.

Everyone's done it at least once. You're sure **your code compiles**; you've tested it over and over again on your machine and committed it into the repository. But somewhere between your machine and that black box they call a server, *someone* must have changed your code. The unlucky soul who does the next checkout is about to have a bad morning sorting out **what used to be working code**. In this chapter we'll talk about how to put together a **safety net** to keep the build in working order and you **productive**.

Things will ALWAYS go wrong...

Everyone who's ever done development knows what it's like. It's late, you're on you're eleventh can of Rock Star energy drink, and you still leave out that one ++ operator somewhere. Suddenly, your elegant code goes to pieces...bad news is, you don't ***realize*** you've got a problem.

At least, not until you're demoing the software for your boss. Remember the issues we had with Bob's code in Chapter 6?



BRAIN POWER

What other kinds of things can go wrong on a development project? What about with a small team? Do you have the same problems with a bigger team? Different problems?

Standup meeting

Bob: I did get it compiling! It took me forever to integrate the changes and get everything building again. It's not my fault.

Mark: Yeah, the code compiled; it just didn't work. So he didn't screw up as badly as it looks, really.

Laura: Didn't you test things?

Bob: Well, the code worked fine on my machine. I ran it and everything seemed fine...

Mark: OK, but running your code and doing a quick checkover is not really putting your code to the test.

Laura: Exactly. The functionality of your software is part of your responsibility, not just that the code "seems to work"; that's never going to wash with the customer...

Bob: Well, now that we have a version control server and build tool in place, this shouldn't be a problem anymore. So enough beating up on me, alright?

Mark: Hardly! Our build tool makes sure the code compiles, and we can back out changes with version control, but that doesn't help making sure things work right. Your code compiled; that was never the problem. It's the functionality of the system that got screwed up, and our build tool does nothing for that.

Laura: Yeah, you didn't even realize anything had gone wrong...

There are three ways to look at your system...

Good testing is essential on any software project. If your software doesn't work, it won't get used—and there's a good chance you won't get paid. So before getting into the nitty-gritty of software testing, it's important to step back and remember that different people look at your system from totally different perspectives, or views.

For more on these different types of testing, see Appendix i.

Your users see the system from the outside

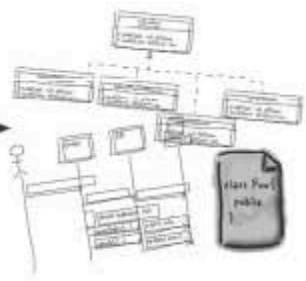
Your users don't see your code, they don't look at the database tables, they don't evaluate your algorithms...and generally *they don't want to*. Your system is a ***black box*** to them; it either does what they asked it to do, or it doesn't. Your users are all about **functionality**.

Testers peek under the covers a little

Testers are a different breed. They're looking for functionality, but they're usually poking underneath to make sure things are really happening the way you said they would. Your system is more of a ***grey box*** to them. Testers are probably looking at the data in your database to make sure things are being cleaned up correctly; they might be checking that ports are closed, network connections dropped, and that memory usage is staying steady.

Developers let it all hang out

Developers are in the weeds. They see good (and sometimes bad) class design, patterns, duplicated code, inconsistencies in how things are represented. The system is wide open to them. If users see a system as a closed black box, developers see it as an open ***white box***. But sometimes because developers see so much detail, it's possible for them to miss broken functionality or make an assumption that a tester or end user might not.

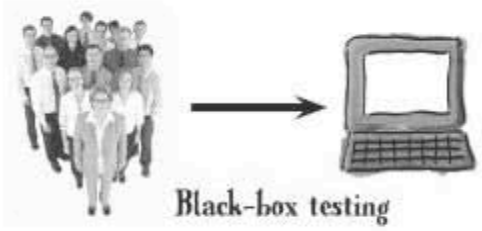


...and you need to consider each of these views

Each view of your system is valid, and you have to test from each of those three perspectives.

Black-box testing focuses on INPUT and OUTPUT

Your users are outside your system. They only see what they put into the system and what comes back out. When you do black-box testing you should look for:



Black-box testing

- ☐ **Functionality**. Hands down, this is the most important black box testing. Does the system do what the user story says it is supposed to do? With black box testing, you don't care if your data is being stored in a text file or a massively parallel clustered database. You just care that the data gets in there like the story says and you get back the results the story says you should.

This isn't "the OrderProcessor class can handle GiftCard objects" functionality; it's about if a customer can buy a drink with their gift card.

- ☐ **User input validation**. Feed your system 3.995 for a dollar amount or -1 for your birthday. If you're writing a web application, put some HTML in your name field or try some SQL. The system better reject those values, and do it in a way that a typical end user can understand.

- ☐ **Output results**. Hand-check numerical values that your system returns. Make sure all of the functional paths have been tested ("if the user enters an invalid ending location, and then clicks "Get Directions"...") It's often helpful to put together a table showing the various inputs you could give the system, and what you'd expect the results to be for each input.

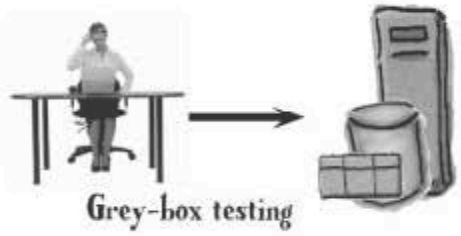
Error conditions are usually the last thing most developers think about, but it's the first thing most customers notice.

- ☐ **State transitions**. Some systems need to move from one state to another according to very specific rules. This is similar to output results, but it's about making sure your system handles moving from state to state like it's supposed to. This is particularly critical if you're implementing some kind of protocol like SMTP, a satellite communications link, or GPS receiver. Again, having a map of the states and what it takes to move the system from one to the other is very useful here.

- ☐ **Boundary cases and off-by-one errors**. You should test your system with a value that's just a little too small or just outside the maximum allowable value. For example, checking month 12 (if your months go from 0–11) or month 13 will let you know if you've got things just right, or if someone slipped up and forgot about zero-based arrays.

Customers don't usually make huge mistakes—they make little typos, and those are the things you're testing for here.

Grey-box testing gets you CLOSER to the code


Grey-box testing

Black-box testing works great for a lot of applications, but there are situations where you need more. Sometimes you just can't get the results out of a system easily without looking inside, at least a little. This is particularly true with a lot of web applications, where the web interface just moves data around in a database. You've got to deal with the database code ***as well as*** the web interface itself.

Grey-box testing is like black-box testing...but you can peek

When doing grey box testing, you're generally looking for the same things as black box testing, but you can dig around a little to make sure the system works as it's supposed to below the surface. Use grey box testing for things like:

But be careful of logging confidential information to unsecured places, you won't make the right sorts of friends that way...

- ☐ **Verifying auditing and logging**. When important data (or money) is on the line, there's usually a lot of auditing and logging going on inside a system. This information isn't usually available through the normal user interface, either. You might need to use a log viewing tool or auditing report, or maybe just query some database tables directly.

- ☐ **Data destined for other systems**. If you're building a system that sends information to another system at a later time (say an order for 50 copies of *Head First Software Development*), you should check the output format and data you're sending to the other systems...and that means looking underneath what's exposed by the system.

- ☐ **System-added information**. It's common for applications to create checksums or hashes of data to make sure things are stored correctly (or securely). You should hand-check these. Make sure system-generated timestamps are being created in the right time zone and stored with the right data.

- ☐ **Scraps left laying around**. It's so easy as a developer to miss doing cleanup after a system is done with data. This can be a security risk as well as a resource leak. Make sure data is really deleted if it's supposed to be, and make sure it isn't deleted if it's not. Check that the system isn't leaking memory while it's running. Look for things that might leave scraps of files or registry entries after they should have been cleaned up. Verify that uninstalling your application leaves the system clean.

Exercise

Below is a user story from BeatBox Pro. Your job is to write up three ideas for black or grey box tests, and descriptions of what you'd do to implement those tests.

Title: Send a picture to other users

Description: Click on the "Send a Picture" button to send a picture (only JPEG needs to be supported) to the other users. They should have the option to not accept the file. There are no size limits on the file right now.

Priority: 20 **Estimate:** 4

1. Test for... sending a small JPEG to another user

Here's one to get you started

How would you test this? Describe the test case in plain English.

2. Test for...

Think about the different ways the fuctionality in the user story could be tested, like testing when it handles things going wrong...

3. Test for...

Below is a user story from BeatBox Pro. Your job was to write up three ideas for black or grey box tests, and descriptions of what you'd do to implement those tests.

Title: Send a picture to other users

Description: Click on the "Send a Picture" button to send a picture (only JPEG needs to be supported) to the other users. They should have the option to not accept the file. There are no size limits on the file right now.

Priority: 20 **Estimate:** 4

* Here are the three tests we came up with. It's okay if you have three different ones... you'll just have more ideas for actual tests.

1. Test for... sending a small JPEG to another user.

Get two instances of BeatBox Pro running. On the first instance, click the Send Picture button. When the image selection dialog pops up, select SmallImage.jpg and click OK.

Then check and make sure that the second BeatBox displays a Receive Image dialog box. Click OK to accept the image. Check that the image displays correctly.

This is a black-box test. Also notice that we needed some JPEG resources to support the test. That's OK; using sample input is fine. You should version-control those resources, though, for later reuse.

2. Test for... sending an invalid JPEG to another user.

Get two instances of BeatBox Pro running. On the first instance, click the Send Picture button. When the image selection dialog pops up, select InvalidImage.jpg and click OK.

Check that BeatBox shows a dialog telling you that the image is invalid and can't be sent. Confirm that the second BeatBox did not display a Receive Image dialog. Also make sure no exceptions were thrown from either instance.

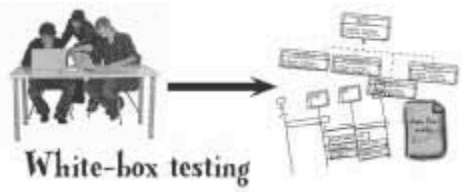
These tests are a little more on the grey side. You need to know how BeatBox Pro should handle these conditions, and where exceptions would be sent if an error occurred.

3. Test for... losing connectivity while transferring an image.

Start two instances of BeatBox Pro. On the first instance, click the Send Picture button. When the image selection dialog pops up, select GiantImage.jpg and click OK.

Check that the second BeatBox shows a Receive Image dialog box and click OK. While this image is transferring (make the image several MB so it will take a while), kill the second BeatBox instance. Check that the first BeatBox displays a dialog saying the transfer failed and that no exceptions were thrown.

White-box testing uses inside knowledge



At the deepest levels of testing, you'll find white box tests. This is where you know exactly what's going on inside the code, and you do your best to make that code break. If you put aside the fact that you have to fix the code when it does break, white-box testing can actually be fun: it becomes a challenge to dig into code and generate problem situations that will cause errors and crashes.

When doing white-box testing you should be familiar with the code you're about to test. You still care about functionality, but you should also be thinking about the fact that method X is going to divide by one of the numbers you're sending in... is that number being checked properly? With white-box testing you're generally looking for:

- ☐ **Testing all the different branches of code**. With white-box testing you should be looking at ***all*** of your code. You can see all of the `if/else`s and all the case and switch statements. What data do you need to send in to get the class you're looking at to run each of those branches?

Most code works great when things are going as expected—the so-called "happy path"—but what about when things go off-track?

- ☐ **Proper error handling**. If you do feed invalid data into a method, are you getting the right error back? Is your code cleaning up after itself nicely by releasing resources like file handles, mutexes, or allocated memory?

Black-box testing looked at error messages, but what about what the code left around when things go wrong? That's for white-box testing to examine.

- ☐ **Working as documented**. If the method claims it's thread-safe, test the method from multiple threads If the documentation says you can pass null in as an argument to a method and you'll then get back a certain set of values, is that what's really going on? If a method claims you need a certain security role to call it, try the method with and without that role.

- ☐ **Proper handling of resource constraints**. If a method tries to grab resources—like memory, disk space, or a network connection—what does the code do if it can't get the resource it needs? Are these problems handled gracefully? Can you write a test to force the code into one of those problematic conditions?

White-box tests tend to be code-on-code

Since white-box tests tend to get up close and personal with the code they're trying to test, it's common to see them written in code and run on a machine rather than exercised by a human. Let's write some code-on-code tests now...

Exercise

Below is the block of code that Bob built for the BeatBox Pro demo (the one that failed spectacularly), and the two user stories that version of the software was focused on. On the next page are three tests that need to pass. How would you test these in code?

These stories have to work in the demo—you have to test for this functionality.

Title: Send a poke to other users

Description: Click on the "Send a Poke" button to send an audible and visual alert to the other members in the chat. The alert should be short and not too annoying—you're just trying to get their attention.

Priority: 20 **Estimate:** 3

Title: Send a picture to other users

Description: Click on the "Send a Picture" button to send a picture (only JPEG needs to be supported) to the other users. They should have the option to not accept the file. There are no size limits on the file right now.

Priority: 20 **Estimate:** 4

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
  try {
    while ((obj = in.readObject()) != null) {
      System.out.println("got an object from server");
      System.out.println(obj.getClass());
      String nameToShow = (String) obj;
      checkboxState = (boolean[]) in.readObject();
      if (nameToShow.equals(PICTURE_START_SEQUENCE)) {
        receiveJPEG();
      }
      else {
        otherSeqsMap.put(nameToShow, checkboxState);
        listVector.add(nameToShow);
        incomingList.setListData(listVector);
        // now reset the sequence to be this
      }
    } // close while
  } catch (Exception ex) {
    ex.printStackTrace();
  }
 } // close run
}
```

Remember that Bob overwrote the code to handle the POKE_START_SEQUENCE command.

How could you test this code to make sure it works, even if another problem comes up?

Definitely pseudocode...if you need a resource, assume you can get it. This is just the basic code-level steps you'd need.

1. Test for... a picture start sequence to test picture functionality.

establish a new network connection
send the PICTURE_START_SEQUENCE
send over an empty array of check boxes (no audio)
send the picture data
verify picture data received and displayed properly

Here's what you need to test

This one is done for you to give you an idea of the pseudocode to use to describe a test.

2. Test for... a poke start sequence to test for poke functionality.

establish a new network connection

What would this test do? This should be pseudocode. What code are you going to have to write to implement this test?

3. Test for... a normal text message that's sent to all clients.

establish a new network connection

Below is the block of code that Bob built for the demo (the one that failed spectacularly), and the two user stories this version of the software was focused on. Your job was to figure out how to white-box-test for at least three problem situations.

test for a picture start sequence to test picture functionality.

establish a new network connection
send the PICTURE_START_SEQUENCE
send over an empty array of checkboxes (no audio)
send the picture data
verify.picture.data.received.and.displayed.properly.

This is to test the basic picture functionality, since it was one of our new stories, but digs into the code involved.

This is more in-depth than just using the GUI: you're really testing specific methods, with specific inputs, to make sure the result is what's expected.

test for a poke start sequence to test for poke functionality.

establish a new network connection
send the POKE_START_SEQUENCE
send over an empty array of check boxes (no audio)
verify alert sound is heard and the alert message is shown.

With a test to check the POKE_START_SEQUENCE, you can see if it fails before showing it the customer, and avoid any surprises.

You'll probably need to verify this works by watching a running chat client—that's okay, use whatever you need to test properly.

This one is based on the other story, and is a lot like the picture test.

test for a normal text message is sent to all clients.

establish a new network connection
send the message, "Test message"
send an array of valid checkbox options
verify the test message was received by all clients and the checkboxes.are.updated.to.match.the.array.values.

Don't forget to test stuff that should still be working! This is just as important as testing new functionality.

There are lots more tests you could have come up with—things like testing that clicking on one of the messages retrieves the checkboxes correctly, and testing for failure conditions. What happens if too many checkbox values are sent in an array? Or too few? See how many ways you can break BeatBox Pro.

Coming up with tests is YOUR job.

There are lots of good frameworks out there, but they ***run*** your tests; they don't write them for you.* A testing frameworks is really just a collection of tools that help you express your tests. Even though that makes them really useful, there are a few things you still need to keep in mind:

First, **you still need to figure out what you have to test**. Figuring out what to test and how you express that test are usually two different things. Regardless of your framework, you need to think about functional testing, performance testing, boundary or edge cases, race conditions, security risks, valid data, invalid data, etc.

Next, **your choice of testing frameworks is almost certainly going to impact how you test**. That's not always a bad thing, but don't forget about it. This might mean you need more than one way to test your software. For example, if you decide to use a code-level testing framework for your desktop application, you're still leaving yourself open for bugs in your GUI, so you'll probably want something to test that, too. Another great example: say you're writing a 3-D game. Testing the backend code isn't too hard, but making sure that the game renders correctly and people can't walk through walls or fall through small cracks in your world...well, that's a mess, and no framework can generate those tests for you.

* Actually, some frameworks can generate tests for you, but they have very specific goals in mind. Security frameworks are a common example: the framework can throw tons of common security errors at your software and see what happens. But this doesn't replace real application testing to make sure the system does what you think it does (and what the customer actually wants it to do).

Hanging your tests on a framework

We're talking about frameworks, but what does that really mean? The obvious way to test is to have someone **use your application**. But, if we can automate our tests we can ~~get paid while the computer tests our stuff~~ be more effective and know that our tests are run exactly the same way each time. That's important, because consistency in how a test is run isn't something humans are very good at.

Testing EVERYTHING with one step

Well, one command actually

There are lots more advantages to automating your tests. As well as not requiring you to sit there and manually run the tests yourself, you also build up a **library of tests** that are all run at the same time to test your software completely every time you run the **test suite**:

1 Build up a suite of tests

As your software grows so will the tests that need to be applied to it. At first, this might seem a little scary, especially if you're running tests by hand. Large software systems can have literally thousands of tests that take days of developer time to run. If you automate your tests you can collect all the tests for your software into one library and then run those tests at will, without having to rely on having somebody, probably a poor test engineer who looked at you wrong, running those tests manually for a day or so.

2 Run all your tests with one command

Once you have a suite of tests that can be run automatically in a framework, the next step is to build that set of tests such that they can all be run with just one command. The easier a test suite is to run, the more often it will actually be used and that can only mean that your software quality will improve. Any new tests are simply added to the test suite, and bang, everyone gets the benefit of the test you have written.

3 Get regression testing for free

The big advantage of creating a one-command suite of tests that you continually add to as you add more code to your software is that you get **regression testing** for free. Detecting when a new change that you've made to your software has actually introduced bugs in the older code, called **software regression**, is a danger for any developer working with old or inherited code. The best way to deal with the threat of regression problems is to not only run your own tests for your newly added code, but to run all the older tests as well.

Now, because you'll be adding your new tests into your test suite, you'll get this for free. All you have to do is add your new tests to the existing test suite and kick things off with one command—you'll have regression tested your changes.

Of course, this relies on the existing code base having a suite of tests available for you to extend. Check out Chapter 10 for what to do when that isn't the case.

Tailor your test suites to suit the occasion

It's unfortunately true that large unit test suites become ungainly and, therefore, tend to get used less. One technique is to break out fast and slow tests so that a developer can run all the fast tests often while they are changing and adding code, but only run the full suite when they think they need to.

What tests fall into the fast or slow categories is really up to your particular project, and which category specific tests fall into can change depending on the development work that you are doing. For example, if you have barely-ever-changes code that takes a long time to test, then that would be a good candidate for the slow test suite. However if you were working on code that might well impact the barely-ever-changes code, then you might consider moving its tests into the fast test suite while you are working those changes.

Let's try it out with a popular free testing framework for Java, called JUnit.

To download the JUnit framework, go to http://www.junit.org

You can also speed up slow tests using mocks; see Chapter 8 for more on those.

there are no Dumb Questions

Q: So how often should we run our entire test suite?

A: This is really up to you and your team. If you're happy with running your full test suite once a day, and know that any regression bugs will only be caught once a day, then that's fine. However, we'd still recommend you have a set of tests that can be run much more frequently.

Keep the time it takes to run your tests as short as possible. The longer a test suite takes to run, the less often it is likely to be run!

Automate your tests with a testing framework

Let's take a simple test case and automate it using JUnit. JUnit provides common resources and behaviors you need for your tests, and then invokes each of your tests, one at a time. JUnit gives you a nice GUI to see your tests run, as well, but that's really a small thing compared to the power of automating your tests.

JUnit also has a text-based test runner and plug-ins for most popular IDEs.

You've got to import the JUnit classes.

```
package headfirst.sd.chapter7;
import java.io.*;
import java.net.Socket;
import org.junit.*;

public class TestRemoteReader {
    private Socket mTestSocket;
    private ObjectOutputStream mOutStream;
    private ObjectInputStream mInStream;

    public static final boolean[] EMPTY_CHECKBOXES = new boolean[256];

    @Before
    public void setUp() throws IOException {
            mTestSocket = new Socket("127.0.0.1", 4242);
            mOutStream =
              new ObjectOutputStream(mTestSocket.getOutputStream());
            mInStream =
              new ObjectInputStream(mTestSocket.getInputStream());
    }

    @After
    public void tearDown() throws IOException {
            mTestSocket.close();
            mOutStream = null;
            mInStream = null;
            mTestSocket = null;
    }

      @Test
      public void testNormalMessage()throws IOException {
            boolean[] checkboxState = new boolean[256];
            checkboxState[0] = true;
            checkboxState[5] = true;
            checkboxState[19] = true;
            mOutStream.writeObject("This is a test message!");
            mOutStream.writeObject(checkboxState);
      }
}
```

These are objects used in several of the test cases.

Here's a static final of empty checkboxes that can be used in several different tests.

JUnit calls setUp() before each test is run, so here's where to initialize variables used in the test methods.

tearDown() is for cleaning up. JUnit calls this method when each test is finished.

Since these are annotated with @Before and @After, they'll get called by JUnit before and after each test.

Here's an actual test. You annotate it with @Test so JUnit knows it's a test and can run it. The method just sends a test message and a checkboxState.

You can use mOutStream because it was set up in the setup() method that JUnit will already have called.

Use your framework to run your tests

Invoke the JUnit test runner, `org.junit.runner.JUnitCore`. The only information you need to give the runner is which test class to run: `headfirst.sd.chapter7.TestRemoteReader`. The framework handles running each test in that class::

Don't forget to start the MusicServer and a copy of the BeatBox Pro. JUnit won't take care of that for you, unless you add code for that into setUp().

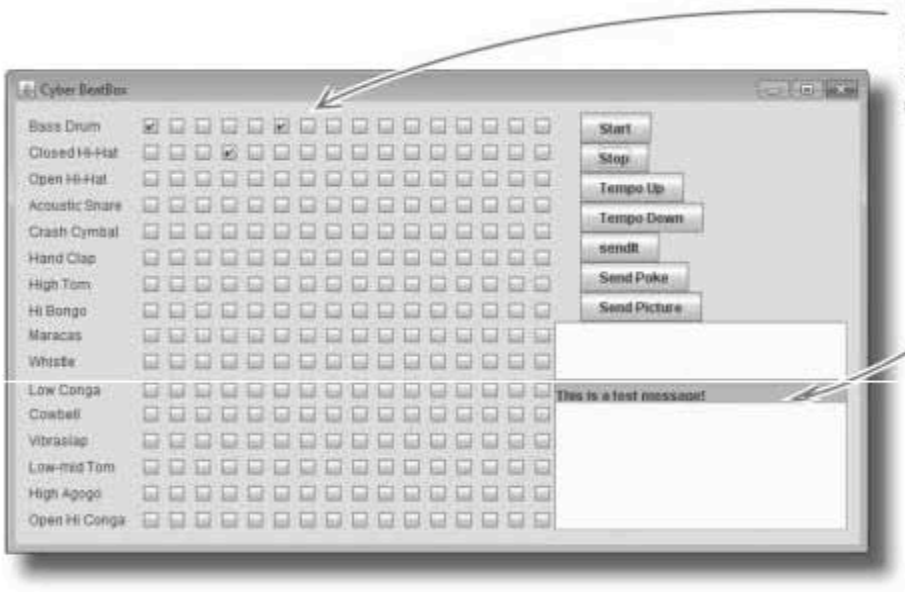
Don't forget to put junit.jar in your classpath.

```
File Edit Window Help
hfsd> java -cp junit.jar;. org.junit.runner.JUnitCore
                    headfirst.sd.chapter7.TestRemoteReader
JUnit version 4.3.1
.
Time: 0.321

OK (1 test)

hfsd>
```

JUnit will print a dot for each test it ran. Since this class has only one test, you get a single dot.

"OK" is JUnit's understated way of saying all the tests ran.

And here's what BeatBox Pro looks like after the test has run. Checkmarks are where they're supposed to be and the test message is in the log.



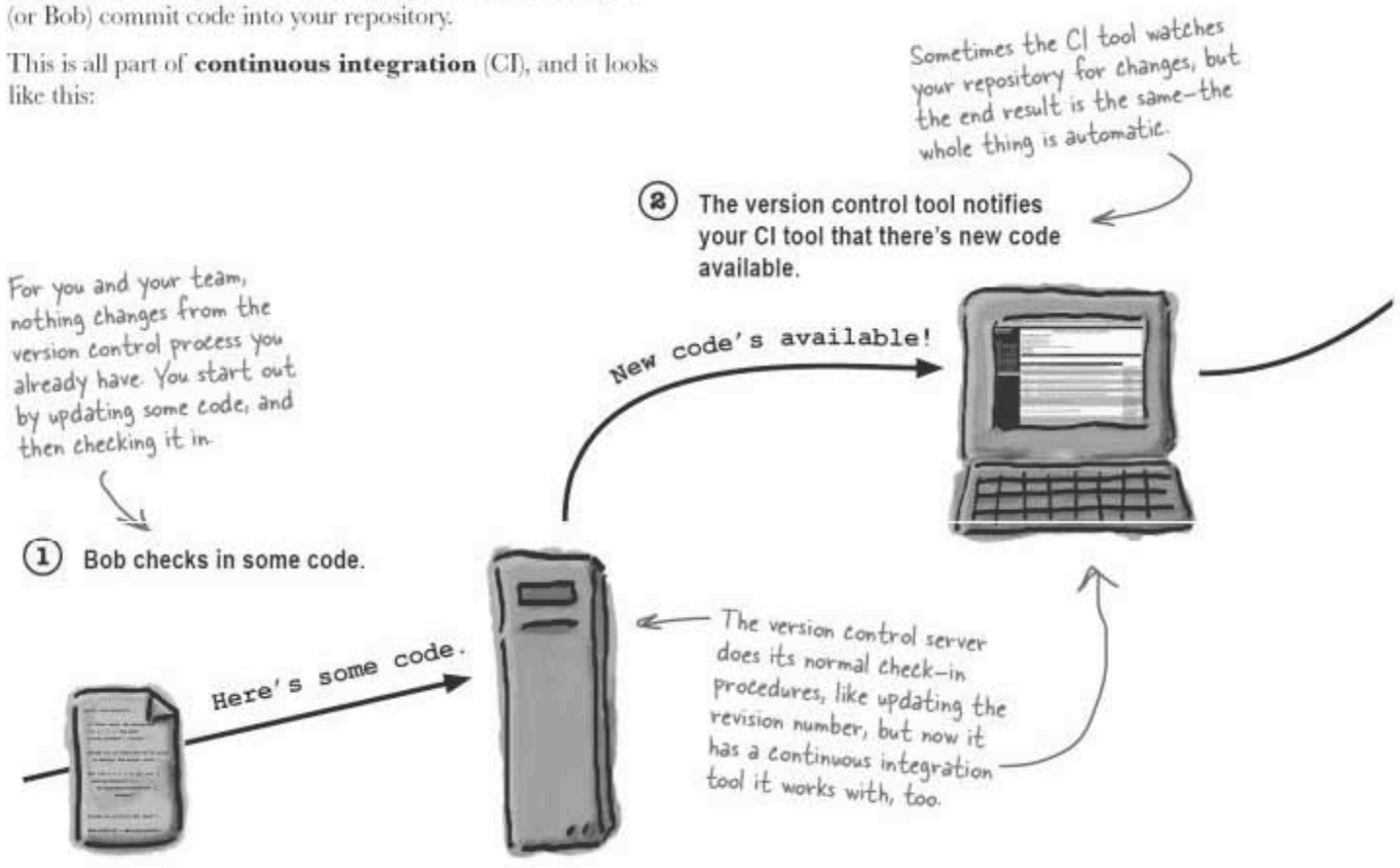
With a framework in place, you can easily add the other tests from page 246. Just add more test methods and annotate them with `@Test`. You can then run your test classes and watch the results.

Wouldn't it be dreamy if there was a tool that ran all my tests for me, every time I checked in code, so I wouldn't be embarrassed in front of my team?

Continuous integration tools run your tests when you check in your code

We've already got a version control tool that keeps track of our code, and now we've got a set of automated tests. We just need a way to tie these two systems together. There are version control tools (or applications that integrate with version control tools) that will compile your code, run your automated tests, and even display and mail out reports—as soon as you (or Bob) commit code into your repository.

This is all part of **continuous integration** (CI), and it looks like this:



The great thing about version control and CI is that they happen without you having to do anything—it's all going on "under the hood."

Continuous integration and build tools are two more processes that improves communication amongst your team.

③ **The CI tool checks out the new code, compiles it, and runs all your tests. Most build tools create web pages and emails to let everyone know how the builds are going.**

This particular build tool is called CruiseControl, but there are lots of similar products out there.

there are no Dumb Questions

Q: Does CI have to build and test my code every time I check it in? My project is so large that could really slow things down.

A: No, defnitely not. Although building and running your tests every time you commit changes to version control is a good practice, sometimes it's not entirely practical. If you have a really large set of tests that use significant computing resources, you might want to schedule things a bit differently.

Continuous integration wraps version control, compilation, and testing into a single repeatable process.

At the wheel of CI with CruiseControl

The three main jobs of a CI tool are to get a version of the code from your repository, build that code, and then run a suite of tests against it. To give you a flavor of how CI is set up, let's take a look at how that works in CruiseControl:

1 **Add your JUnit test suite to your Ant build**

Before you build your CruiseControl project, you need to add your JUnit tests into your Ant build file.

You last saw Ant in Chapter 6.5.

```
<target name="test" depends="compile">
  <junit>
    <classpath refid="classpath.test" />
    <formatter type="brief" usefile="false" />
    <batchtest>
      <fileset dir="${tst-dir}" includes="**/Test*.class" />
    </batchtest>
  </junit>
</target>

<target name="all" depends="test" />
```

A new target called "test" that depends on the "compile" target having finished successfully

Here's where the magic happens. All of the classes in your project that begin with the word "Test" are automatically executed as JUnit tests. No need for you to specify each one individually.

The "all" target is just a nicer way of saying "compile, build, and test everything."

2 **Create your CruiseControl project**

The next step is to create a CruiseControl project and begin to define your build and test process.

```
<cruisecontrol>
  <project name="BeatBox" buildafterfailed="true">

    <!-- This is where the rest of your project configuration will go -->

  </project>
</cruisecontrol>
```

The project tag bounds all of your project's configuration.

In CruiseControl, your project is described using an XML document, much the same as in Ant, except this script describes what is going be done, and when.

3 Check to see if there have been any changes in the repository

Inside your CruiseControl project you can describe where to get your code from and then what to do with it. In this case, code changes are grabbed from your subversion repository. If the code has changed, then a full build is run; otherwise the scheduled build is skipped.

The "modificationset" tells the repository to check against the local copy to see if it actually needs to build changes in or not

Here you declare what local copy and remote repository to check against for changes

```
<modificationset quietperiod="10">
  <svn LocalWorkingCopy="hfsd/chapter7/cc"
   RepositoryLocation="file:///c:/Users/Developer/Desktop/SVNRepo/BeatBox/trunk"/>
</modificationset>
```

4 Schedule the build

Finally, you describe how often you want your continuous integration build to take place. In CruiseControl this is done with the `schedule` tag, inside of which you describe the type of build that you want to perform.

Schedules the build to occur every 60 minutes

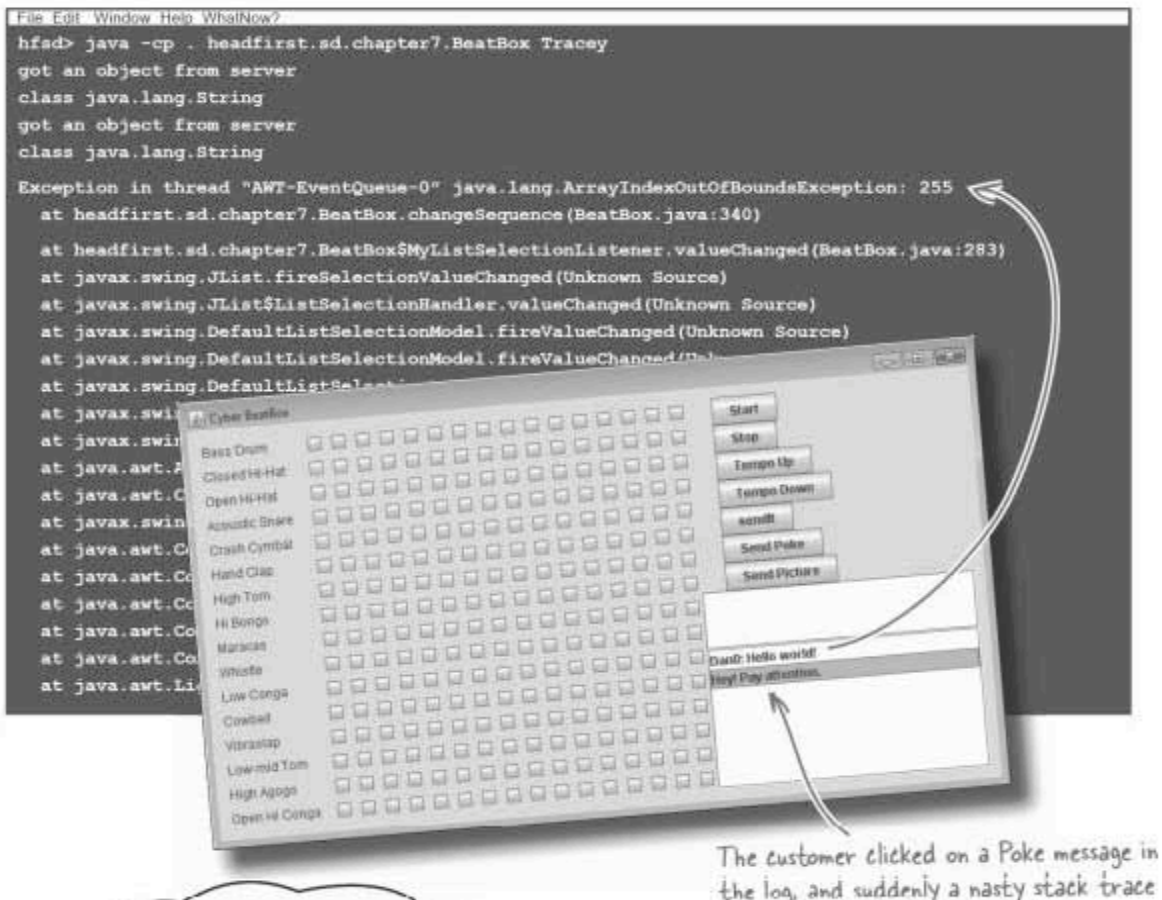
Here, you plug in your Ant build script

Building the "all" target

```
<schedule interval="60">
  <ant antworkingdir="hfsd/chapter7/cc"
       buildfile="build.xml"
       uselogger="true"
       usedebug="true"
       target="all"/>
</schedule>
```

Testing guarantees things will work... right?

Version control, CI, test frameworks, build tools...you've come a long way since it was you and your college buddies hacking away on laptops in your garage. With all your testing, you should be confident showing the customer what you've built:



The customer clicked on a Poke message in the log, and suddenly a nasty stack trace spit out onto the console window.

This is really starting to get old... can't you get ***anything*** right?

Here's the code we changed in Chapter 6. The bug has to be related to this stuff somewhere. Find the bug that bit us this time.

```
public void buildGUI() {
  // code from buildGUI
  JButton sendIt = new JButton("sendIt");
  sendIt.addActionListener(new MySendListener());
  buttonBox.add(sendIt);
  JButton sendPoke = new JButton("Send Poke");
  sendPoke.addActionListener(new MyPokeListener());
  buttonBox.add(sendPoke);
  userMessage = new JTextField();
  buttonBox.add(userMessage);
  // more code in buildGUI()
}

public class MyPokeListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    // We'll create an empty state array here
    boolean[] checkboxState = new boolean[255];
    try {
      out.writeObject(POKE_START_SEQUENCE);
      out.writeObject(checkboxState);
    } catch (Exception ex) {
        System.out.println("Failed to poke!"); }
    }
  }
  // other code in BeatBoxFinal.java
}
```

Here's the code we modified from BeatBox.java's buildGUI() method.

This inner class is from BeatBox.java, too.

What went wrong in this code? .....

.....

Why didn't our tests catch this? .....

.....

What would you do differently? .....

.....

Here's the code we changed in Chapter 6. The bug has to be related to this stuff somewhere. Find the bug that bit us this time.

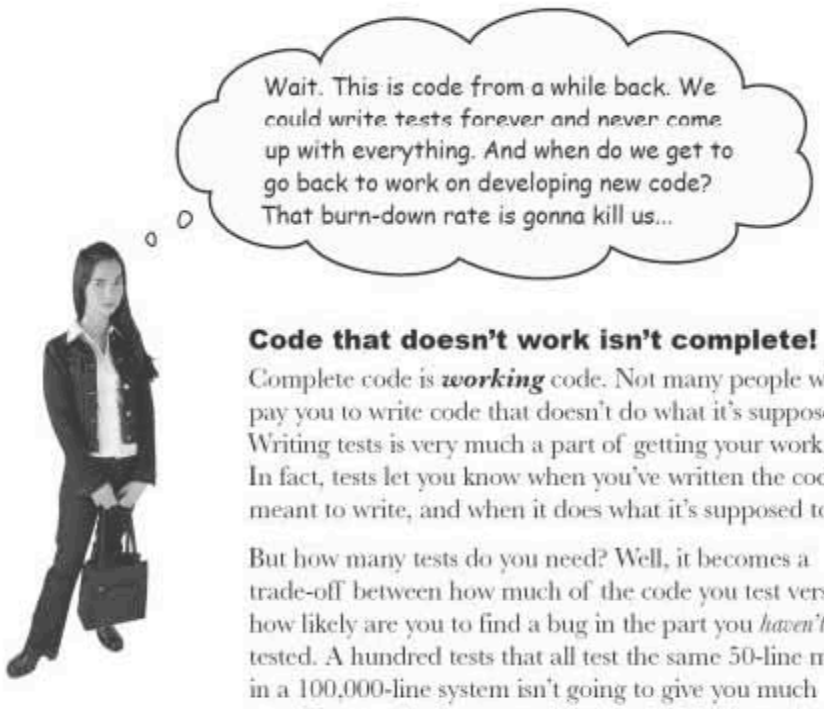
```
public void buildGUI() {
  // code from buildGUI
  JButton sendIt = new JButton("sendIt");
  sendIt.addActionListener(new MySendListener());
  buttonBox.add(sendIt);
  JButton sendPoke = new JButton("Send Poke");
  sendPoke.addActionListener(new MyPokeListener());
  buttonBox.add(sendPoke);
  userMessage = new JTextField();
  buttonBox.add(userMessage);
  // more code in buildGUI()
}

public class MyPokeListener implements ActionListener {
  public void actionPerformed(ActionEvent a) {
    // We'll create an empty state array here
    boolean[] checkboxState = new boolean[255];
    try {
      out.writeObject(POKE_START_SEQUENCE);
      out.writeObject(checkboxState);
    } catch (Exception ex) {
        System.out.println("Failed to poke!"); }
    }
  }
  // other code in BeatBoxFinal.java
}
```

Here's the bug. We create an array of 255 booleans instead of 256.

What went wrong in this code? When we send the dummy array of checkboxes, we're off by one—we only send 255 checkboxes, and it should be 256 (16x16).

Why didn't our tests catch this? Our tests sent valid arrays to our receiver code, but we didn't really test the GUI side of the application.

What would you do differently? We need a way to test more of our code. We should add a test that will catch this. (But what else are we missing?)



Code that doesn't work isn't complete!

Complete code is ***working*** code. Not many people will pay you to write code that doesn't do what it's supposed to. Writing tests is very much a part of getting your work done. In fact, tests let you know when you've written the code you meant to write, and when it does what it's supposed to do.

But how many tests do you need? Well, it becomes a trade-off between how much of the code you test versus how likely are you to find a bug in the part you *haven't* tested. A hundred tests that all test the same 50-line method in a 100,000-line system isn't going to give you much confidence—that leaves a whopping 99,950 lines of untested code, no matter how many tests you've written.

Instead of talking about number of tests, it's better to think about **code coverage**: what percentage of your code are your tests actually testing?

Your code's not finished until it passes its tests.

You should be writing your tests to make sure your code does what it's supposed to. If you don't have a test for a certain piece of functionality, how do you know your code really implements that functionality? And if you do have a test and it's not passing, your code doesn't work.

Tools are your friends.

Tools and frameworks can't do your work for you, but they can make it easier for you to get to your work—and figure out what you should be working on. Code coverage is no different.

Sharpen your pencil

Below is some code from the BeatBox Pro application. Your job is to come up with tests to get 100% coverage on this code... or as close to it as you can get.

This is the code that handles the picture and poke sequences, as well as normal messages.

```
public class RemoteReader implements Runnable {
  boolean[] checkboxState = null;
  String nameToShow = null;
  Object obj = null;

  public void run() {
    try {
      while ((obj = in.readObject()) != null) {
        System.out.println("got an object from server");
        System.out.println(obj.getClass());
        String nameToShow = (String) obj;
        checkboxState = (boolean[]) in.readObject();

        if (nameToShow.equals(PICTURE_START_SEQUENCE)) {
          receiveJPEG();
        } else {
         if (nameToShow.equals(POKE_START_SEQUENCE)) {
           playPoke();
           nameToShow = "Hey! Pay attention.";
         }

         otherSeqsMap.put(nameToShow, checkboxState);
         listVector.add(nameToShow);
         incomingList.setListData(listVector);
         // now reset the sequence to be this
        }
      } // close while
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  } // close run
} // close inner class
```

Circle any code that your tests don't cover.

1. Write a test to exercise this section of the code (pseudocode is fine).

2. Write a test to exercise this section of the code.

3. Write a test to exercise this section of the code.

4. Did we get 100% coverage? What else would you test? How?

Some of these tests may test more than just the section of code bracketed—write notes indicating what else your tests exercise.

Sharpen your pencil Solution

Below is some code from the BeatBox Pro application. Your job was to come up with tests to get 100% coverage on this code...or as close to it as you can get.

```
public class RemoteReader implem
  boolean[] checkboxState = null
  String nameToShow = null;
  Object obj = null;

  public void run() {
    try {
      while ((obj = in.readObject
        System.out.println("got an object from server");
        System.out.println(obj.getClass());
        String nameToShow = (String)
        checkboxState = (boolean[]) i

        if (nameToShow.equals(PICTURE
          receiveJPEG();
        } else {
          if (nameToShow.equals(POKE_START_SEQUENCE)) {
            playPoke();
            nameToShow = "Hey! Pay attent
          }

          otherSeqsMap.put(nameToShow, c
          listVector.add(nameToShow);
          incomingList.setListData(listVector);
          // now reset the sequence to be this
        }
      } // close while
    } catch (Exception ex) {
      ex.printStackTrace();
    }
  } // close run
} // close inner class
```

All three of these tests cover the code before the if statement

```
1
@Test
public void testNormalMessage()throws IOException {
  boolean[] checkboxState = new boolean[256];
  checkboxState[0] = true;
  checkboxState[5] = true;
  checkboxState[19] = true;
  mOutStream.writeObject("This is a test message!");
  mOutStream.writeObject(checkboxState);
}
```

```
2
@Test
public void testPictureMessage()throws IOException {
  mOutStream.writeObject(PICTURE_START_SEQUENCE);
  mOutStream.writeObject(EMPTY_CHECKBOXES);
  sendJPEG(TEST_JPEG_FILENAME);
}
```

```
3
@Test
public void testPoke()throws IOException {
  mOutStream.writeObject(POKE_START_SEQUENCE);
  mOutStream.writeObject(EMPTY_CHECKBOXES);
}
```

This test exercises multiple chunks of code. In fact, most tests aren't isolated to just a few lines, even though it might be the only test that covers those few lines.

4 Did we get 100% coverage? What else would you test? How?

We didn't test the exception-handling code, so we'd need to create exceptional situations. We also didn't test the GUI at all—that would take someone playing with the interface.

Standup meeting

Mark: No, I don't think so; running every method doesn't mean every ***line*** of each method will run. We need to have different kinds of tests to get to all the different error conditions and branches.

Laura: Wow...so I guess every variation of every method should have a separate test?

Bob: But how are we going to do all that? We'll have to make up all kinds of bogus data to get every weird error condition. That could take forever...

Mark: And that's not all. We've got to try things like pulling the network plug at some point to test what happens if the network goes down and I/O problems crop up.

Bob: You don't think that's going a little too far?

Mark: Well, if we want to catch all of the corner cases and every bit of exception handling...

Laura: But a lot of that stuff never really happens...

Bob: Then why did I bother to write all that exception-handling code? I've got all kinds of logging and reconnection code in my methods. Now you're saying I didn't need to write that?

Mark: You did, but—

Laura: This is impossible!

Testing all your code means testing EVERY BRANCH

Some of the easiest areas to miss are methods or code that have lots of branches. Suppose you've got login code like this:

```
public class ComplexCode {
  public class UserCredentials {
    private String mToken;

    UserCredentials(String token) {
      mToken = token;
    }
    public String getUserToken() { return mToken; }
  }

  public UserCredentials login(String userId, String password) {
    if (userId == null) {
      throw new IllegalArgumentException("userId cannot be null");
    }
    if (password == null) {
      throw new IllegalArgumentException("password cannot be null");
    }
    User user = findUserByIdAndPassword(userId, password);
    if (user != null) {
      return new UserCredentials(generateToken(userId, password,
                  Calendar.getInstance().getTimeInMillis()));
    }
    throw new RuntimeException("Can't find user: " + userId);
  }

  private User findUserByIdAndPassword(String userId, String password) {
    // code here only used by class internals
  }

  private String generateToken(String userId, String password,
                               long nonce) {
    // utility method used only by this class
  }
}
```

You'd probably only need one test case for all of the UserCredential code, since there's no behavior, just data to access and set.

You'll need lots of tests for this method. One with a valid username and password...

...one where the userId is null...

...another where the password is null...

...and one where the username is valid but the password is wrong.

...one where the userId isn't null but isn't a valid ID...

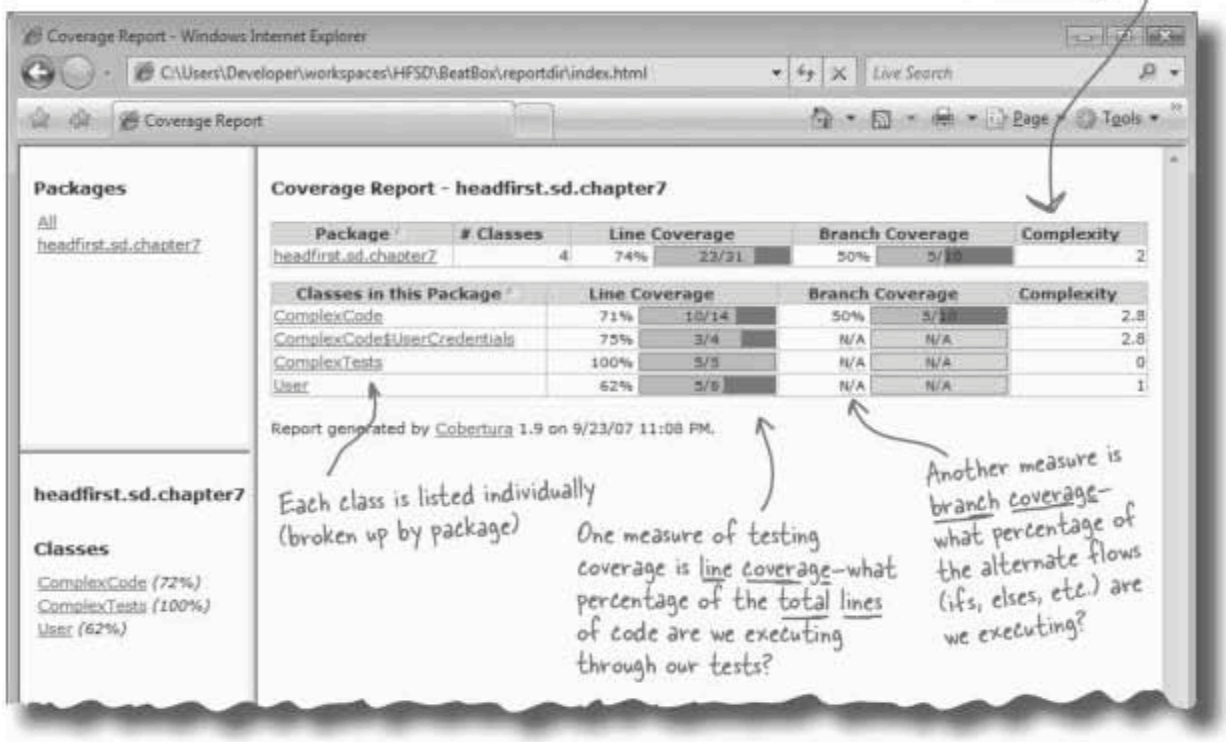
And then there are these private methods...We can't get to these directly.

Use a coverage report to see what's covered

Most coverage tools—especially ones like CruiseControl that integrate with other CI and version control tools—can generate a report telling you how much of your code is covered.

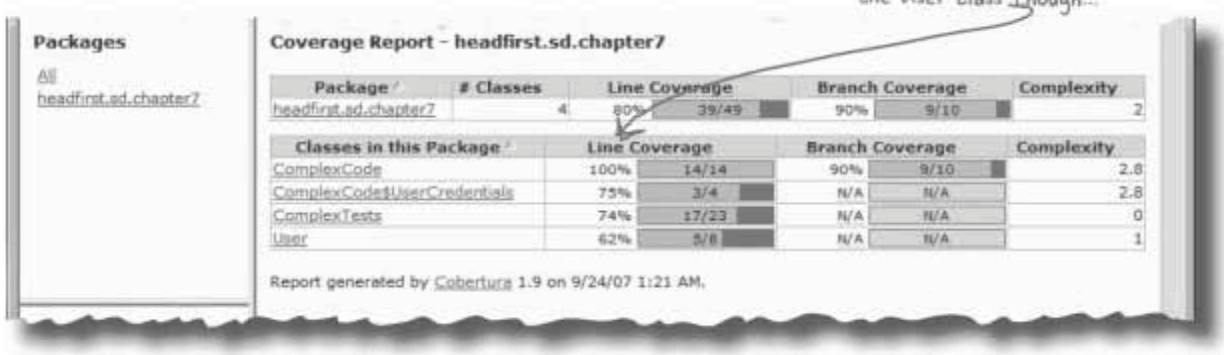
Here's a report for testing the `ComplexCode` class on the last page, and providing a valid username and password:

Code complexity basically tells us how many different paths there are through a given class's code. If there are lots of conditionals (more complicated code), this number will be high.



So the above test manages to test 62% of the `User` class, 71% of the `ComplexCode` class, and 75% of `UserCredentials`. Things get a lot better if you add in all the failure cases described on page 264.

Add in the failure cases and we're in much better shape with the ComplexCode class. Still need work on the User class though...



Good testing takes lots of time.

In general, it's not practical to always hit 100% coverage. You'll get diminishing returns on your testing after a certain point. For most projects, aim for about 85%–90% coverage. More often than not, it's just not possible to tease out that last 10%–15% of coverage. In other cases, it's possible but just far too much work to be worth the trouble.

You should decide on a coverage goal on a per-project, and sometimes even a per-class, basis. Shoot for a certain percentage when you first start, say 80%, and then keep track of the number of bugs found, first using your tests, and then after you release your code. If you get more bugs back after you release your code than you're comfortable with, then increase your coverage requirement by 5% or so.

Keep track of your numbers again. What's the ratio between bugs found by your testing versus bugs found after release? At some point you'll see that increasing your coverage percentage is taking a long time, but not really increasing the number of bugs you find internally. When you hit that point, then back off a little and know you've found a good balance.

there are no Dumb Questions

Q: How do coverage tools work?

A: There are basically three approaches coverage tools can take:

1. They can inspect the code during compilation time
2. They can inspect it after compilation, or
3. They can run in a customized environment (JVM)

Q: We want to try doing coverage analysis on our project, but right now our tests cover hardly anything. How do we get started?

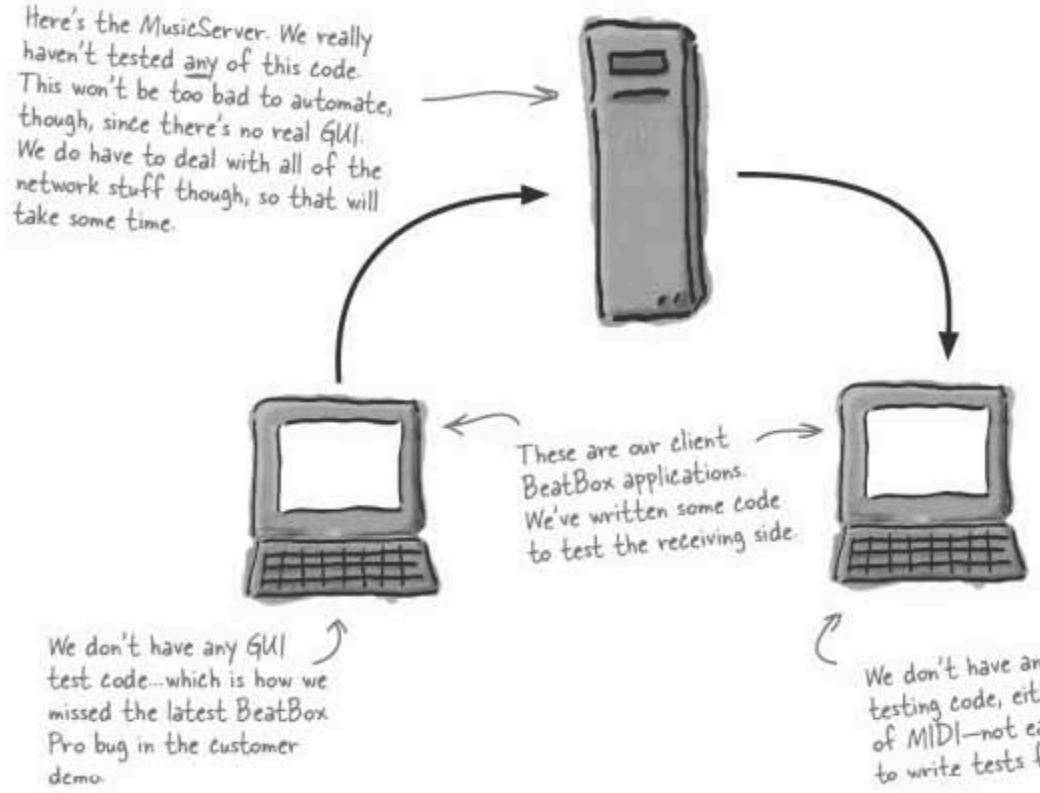
A: Start small. Set your target at 10%. Then when you hit it, celebrate, then bump it to 15%. If you've never done automated testing on your project before, you might find that some parts of your system are really hard to automate. We'll talk more about that in Chapter 8. Get as far as you can, though—some testing is way better than no testing.

Q: Don't you end up with a lot of test code?

A: Absolutely. You'll have a 2-to-1 or 3-to-1 test-to-production code ratio if you're really doing good testing. But finding bugs early is so much easier than having your customer find them. It's more code to maintain, but if your environment is in place, the extra code and effort is generally worth the trade-off. More satisfied customers, more business, and more money!

Getting good coverage isn't always easy...

Now that we've gotten our heads around coverage, let's look back at BeatBox Pro. Now that we know what to look for, there are all kinds of things not being tested:



There are some things that are just inherently hard to test. GUIs actually aren't impossible; there are tools available that can simulate button clicks and keyboard input. Things like audio or 3-D graphics, though, those are tough. The answer? **Get a real person to try things out.** Software tests can't cover all the different variations of an animated game or audio in a music program.

So what about code you just can't seem to reach? Private methods, third-party libraries, or maybe your own code that's abstracted away from the inputs and outputs of your main interface modules? Well, we'll get to that in just a few more pages, in Chapter 8.

And then...enter **test-driven development**.

Sharpen your pencil

Check off all of the things you should do to get good coverage when testing.

- ☐ Test the success cases ("happy paths").
- ☐ Test failure cases.
- ☐ Stage known input data if your system uses a database so you can test various backend problems.
- ☐ Read through the code you're testing.
- ☐ Review your requirements and user stories to see what the system is supposed to do.
- ☐ Test external failure conditions, like network outages or people shutting down their web browsers.
- ☐ Test for security problems like SQL injection or cross-site scripting (XSS).
- ☐ Simulate a disk-full condition.
- ☐ Simulate high-load scenarios.
- ☐ Use different operating systems, platforms, and browsers.

→ Answers on page 272.

Standup meeting

Laura: I really wish we knew all this going in...before we started doing demos with the customer.

Bob: Yeah, I could have run tests on my code, and known I'd screwed up the other user story when I got mine to work. Anything to get us to full coverage...

Mark: Whoa, I'm not sure full coverage is reasonable. You ever heard of the 80/20 rule? Why spend all our time on a tiny bit of the code that probably ***won't*** ever get run?

Bob: Well, I'm going for 100%. I figure with another few days of writing tests, I can get there.

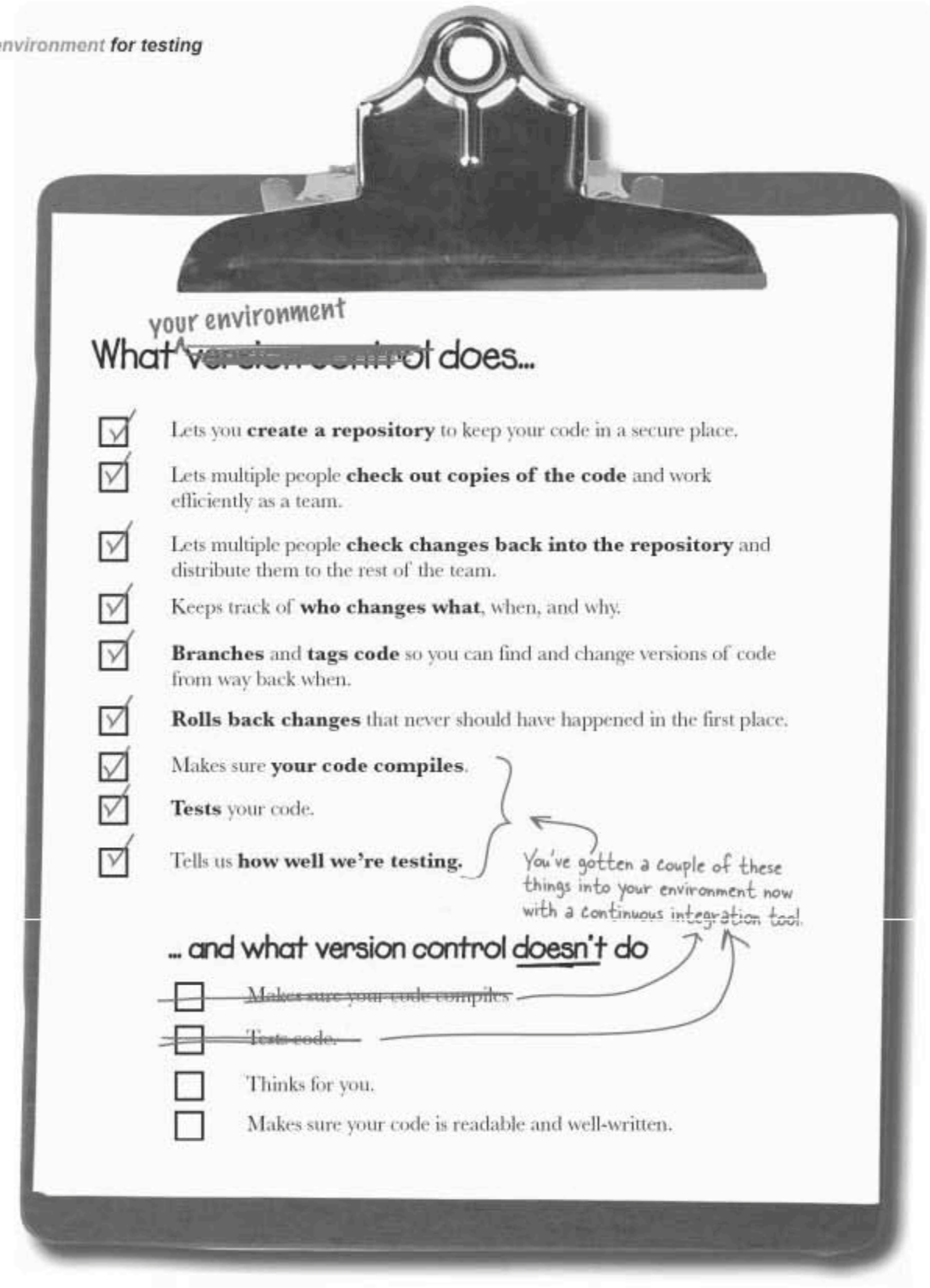
Mark: A few ***days***? We don't have time for that; don't you have a lot of GUI code to work on?

Laura: I agree. But I'm not sure we can even get to 80% coverage: there's a lot of complex code buried pretty deep in the GUI, and I'm not sure how to write tests to get to all of that stuff.

Mark: Hmmm...what about 50%? We could start there, and then add tests for things we think are missing. The coverage report will tell us what we're missing, right?

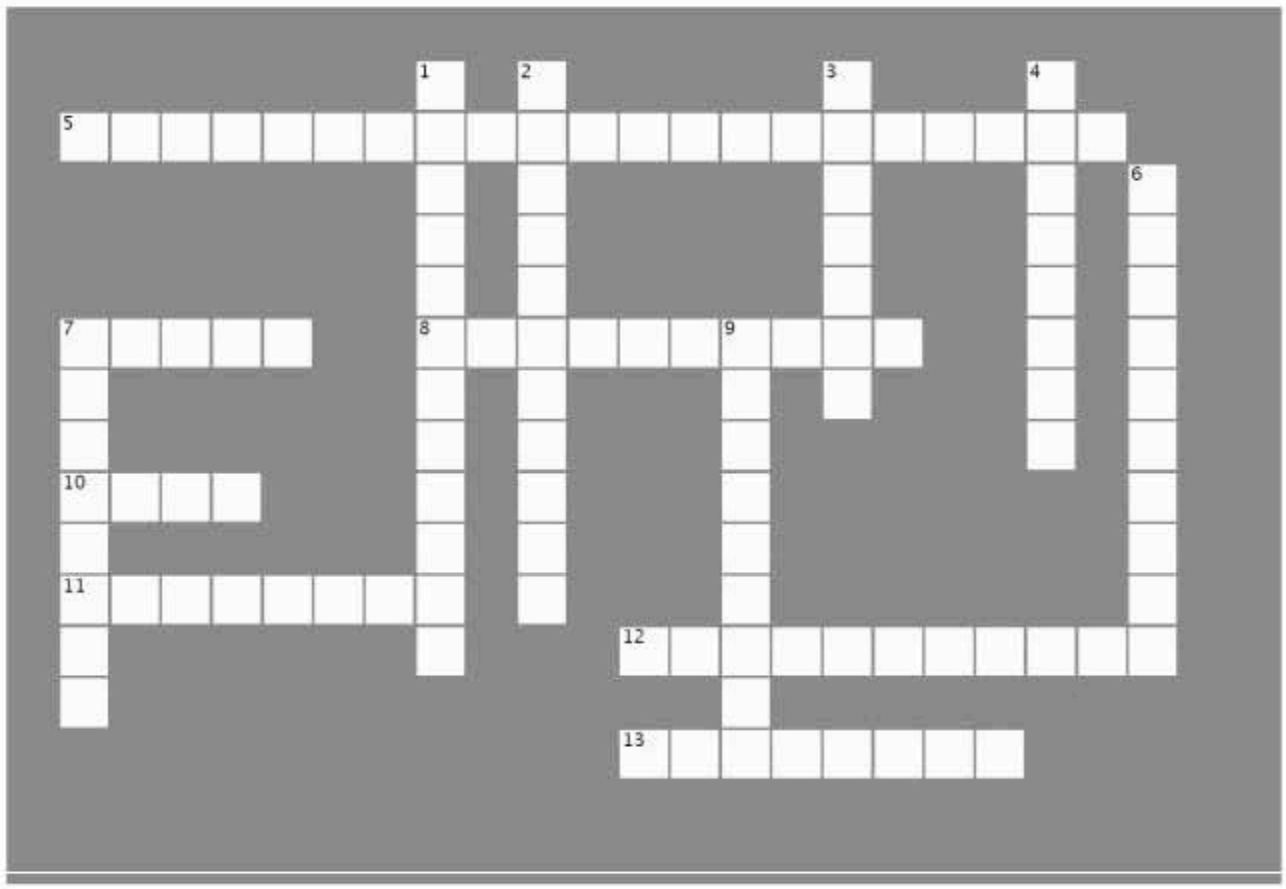
Bob: Yeah, we can look at which methods we're not calling. If we could hit every method, and then test the edge cases on code that's used a lot, that's pretty good...

Laura: Sounds like a plan...You just committed some stuff, right? I'll check the coverage report as soon as CruiseControl finishes its build.


your environment
What ~~version control~~ does...
Lets you **create a repository** to keep your code in a secure place.
Lets multiple people **check out copies of the code** and work efficiently as a team.
Lets multiple people **check changes back into the repository** and distribute them to the rest of the team.
Keeps track of **who changes what**, when, and why.
Branches and **tags code** so you can find and change versions of code from way back when.
Rolls back changes that never should have happened in the first place.
Makes sure **your code compiles**.
Tests your code.
Tells us **how well we're testing.**
You've gotten a couple of these things into your environment now with a continuous integration tool.
... and what version control doesn't do
~~Makes sure your code compiles~~
~~Tests code.~~
Thinks for you.
Makes sure your code is readable and well-written.

Testcross

Take some time to sit back and test the right side of your brain (get it?).



Across

5. The practice of automatically building and testing your code on each commit.
7. This should fail if a test doesn't pass.
8. Instead of running your tests by hand, use .....
10. Coverage tells you how much ..... you're actually testing.
11. When white box testing you want to exercise each of these.
12. Ability to be climbed - or support a lot of users.
13. 3 lines of this to 1 line of production isn't crazy.

Down

1. Just slightly outside the valid range, this case can be bad news.
2. All of your functional testing ties back to these.
3. Peeking under the covers a little, you might check out some DB tables when you use this kind of testing.
4. 85% of this and you're doing ok.
6. Continuous integration watches this to know when things change.
7. Test the system like a user and forget how it works inside.
9. You're done when all your ....

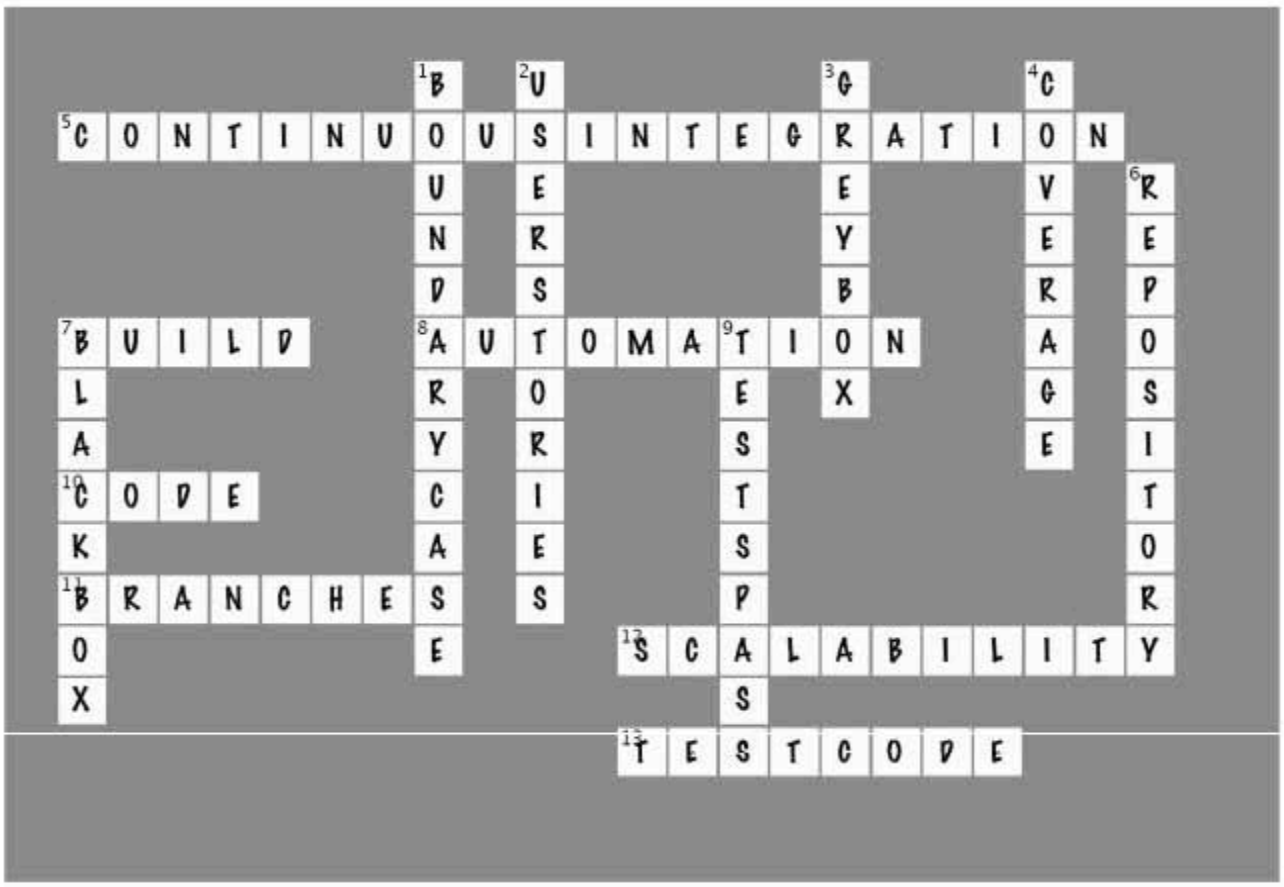
Sharpen your pencil Solution

Check off all of the things you should do to get good coverage when testing.

- [x] Test the success cases ("happy paths").
- [x] Test failure cases.
- [x] Stage known input data if your system uses a database so you can test various backend problems.
- [x] Read through the code you're testing.
- [x] Review your requirements and user stories to see what the system is supposed to do.
- [x] Test external failure conditions, like network outages or people shutting down their web browsers.
- [x] Test for security problems like SQL injection or cross-site scripting (XSS).
- [x] Simulate a disk-full condition.
- [x] Simulate high-load scenarios.
- [x] Use different operating systems, platforms, and browsers.

* Depending on your app, all of these are critical to getting good tests. But, if you're using a coverage tool, you can figure out where you might be missing tests on part of your system.

Testcross Solution



Tools for your Software Development Toolbox

Software Development is all about developing and delivering great software. In this chapter, you learned about several techniques to keep you on track. For a complete list of tools in the book, see Appendix ii.

Development Techniques

There are different views of your system, and you need to test them all

Testing has to account for success cases as well as failure cases

Automate testing whenever possible

Use a continuous integration tool to automate building and testing your code on each commit

Here are some of the key techniques you learned in this chapter...

... and some of the principles behind those techniques.

Development Principles

Testing is a tool to let you know where your project is at all times

Continuous integration gives you confidence that the code in your repository is correct and builds properly

Code coverage is a much better metric of testing effectiveness than test count

BULLET POINTS

- Using **Continuous Integration** tools means something is always watching over the quality of the code in the repository.
- **Automated testing** can be addictive. You still get to write code, so it's fun. And sometimes you break things. Also fun.
- Make the results of your continuous integration builds and coverage reports **public to the team**—the team owns the project and should feel responsible.
- Have your continuous integration tool **fail a build** if an automated test fails. Then have it **email the committer** until they fix it.
- Testing for **overall functionality** is critical to declaring a project as working.

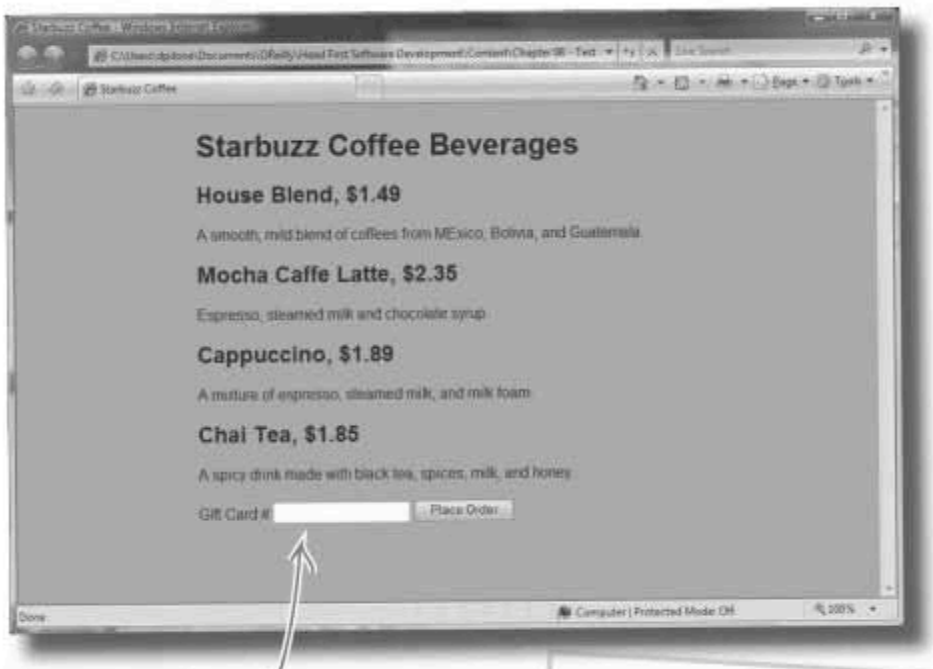
Holding your code accountable

Sometimes it's all about setting expectations. Good code needs to work, everyone knows that. But how do **you know your code works**? Even with unit testing, there are still parts of most code that go untested. But what if testing was a **fundamental part of software development**? What if you did ***everything*** with testing in mind? In this chapter, you'll take what you know about version control, CI, and automated testing and tie it all together into an environment where you can feel **confident** about **fixing bugs**, **refactoring**, and even **reimplementing** parts of your system.

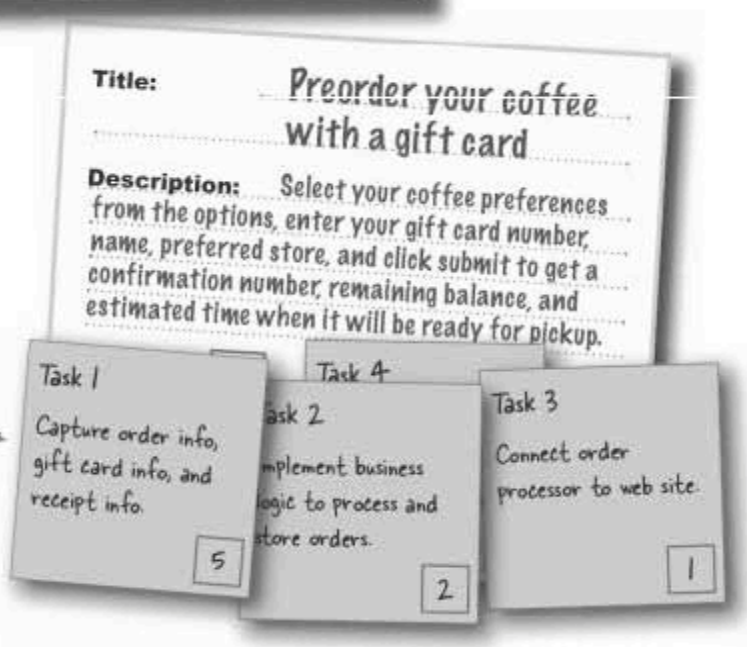
Test FIRST, not last

Instead of trying to retrofit testing onto an existing project, let's look at a project from the ground up using a new technique, **test-driven development**, and write your code with testing in mind right from the start.

Starbuzz Coffee has been selling gift cards for several months, but now they need a way to accept those gift cards as payment for their drinks. Starbuzz already knows how their page should look, so your job is to focus on the design and implementation of the gift card ordering system itself.



Customers can use a gift card to purchase drinks at the new web kiosks in Starbuzz stores



Let's start with this task...

So we're going to test FIRST...

The Starbuzz Gift Cards story is broken down into tasks, so if we're going to test first, we need to begin by looking at our first task, which is capturing information about orders, gift cards, and receipts. Remember, if we jump right into code, we'll end up right back where we did in the last few chapters...

Analyze the task

First break down the task. For this task you'll need to...

Lingo alert: "customer" in these cases refers to shoppers at Starbuzz—in fact, your customer's customer. That's typical language for user stories.

- ☐ **Represent the order information.** You need to capture the customer's name, the drink description, the store number the customer wants to pick up the drink from, and a gift card number.
- ☐ **Represent gift card information.** You need to capture the activation date, the expiration date, and the remaining balance.
- ☐ **Represent receipt information.** You need to capture the confirmation number and the pickup time, as well as the remaining balance on a gift card.

Usually, tasks are just one thing, but the three items in this task are so small, they're easier to treat as a single unit of work.

Write the test BEFORE any other code

We're testing first, remember? That means you have to actually write a test... ***first***. Start with the order information part of the task. Now, using your test framework, you need to write a test for that functionality.

Just like in Chapter 7, you can use any testing framework you want—although an automated framework is easiest to integrate into your version control and CI processes.

Welcome to test-driven development

When you're writing tests before any code, and then letting those tests drive your code, you're using **test-driven development**, or **TDD**. That's just a formal term to describe the process of testing from the outset of development—and writing every line of code specifically as a response to your tests. Turn the page for a lot more on TDD.

Your first test...

The first step in writing a test is to figure out what ***exactly*** it is you should be testing. Since this is testing at a really fine-grained level—**unit testing**—you should start small. What's the ***smallest test you could write*** that uses the order information you've got to store as part of the first task? Well, that's just creating the object itself, right? Here's how to test creating a new `OrderInformation` object:

This is a JUnit test... a single method that tests object creation.

```
package headfirst.sd.chapter8;

import org.junit.*;

public class TestOrderInformation {
  @Test
  public void testCreateOrderInformation() {
    OrderInformation orderInfo = new OrderInformation();
  }
}
```

Keep it as simple as possible: just create a new OrderInformation object

You're exactly right! We're writing tests ***first***, remember? We have **no code**. There's no way this test could (or should) pass the first time through. In fact, this test won't even compile, and that's OK, too. We'll fix it in a minute. The point here is that at first, your test...

...fails miserably.

Unlike pretty much everything else in life, in TDD **you want your tests to fail when you first write them.** The point of a test is to establish a measurable success—and in this case, that measure is a compiling `OrderInformation` object that you can instantiate. And, because you've got a failing test, now it's clear what you have to do to make sure that test passes.

The first rule of effective test-driven development

Rule #1: Your test should always FAIL before you implement any code.

NOW write code to get the test to pass.

You've got a failing test...but that's OK. Before going any further, either writing more tests or working on the task, ***write the simplest code possible to get just this test to pass***. And right now, the test won't even compile!

```
File Edit Window Help
hfsd> javac -cp junit.jar
          headfirst.sd.chapter8.TestOrderInformation.java
TestOrderInformation.java:8: cannot find symbol
symbol  : class OrderInformation
location: class headfirst.sd.chapter8.TestOrderInformation
    OrderInformation orderInfo = new OrderInformation();
    ^
TestOrderInformation.java:8: cannot find symbol
symbol  : class OrderInformation
location: class headfirst.sd.chapter8.TestOrderInformation
    OrderInformation orderInfo = new OrderInformation();
                                     ^
2 errors

hfsd>
```

Running our ~~first~~ test isn't even possible yet; it fails when you try to compile.

Sharpen your pencil

We have a failing test that we need to get to pass. What's the simplest thing you can do to get this test passing?

.....

.....

.....

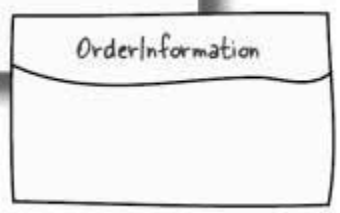
Get your tests to GREEN

The only goal you should have at this point is to get your test to pass. So write *just the code you have to* in order for your test to pass; that's called ***getting your tests to green***.

Green refers to the green bar that JUnit's GUI displays when all tests pass. If any test failed, it displays a red bar.

```
public class OrderInformation {

}
```

Here's the UML for the new class. No attributes, no methods—just an empty class.



Yes, that's it. An empty class. Now try running your test again:

```
File Edit Window Help Classy
hfsd> javac -d bin -cp junit.jar *.java

hfsd> java -cp junit.jar;.\bin org.junit.runner.
JUnitCore headfirst.sd.chapter8.TestOrderInformation
JUnit version 4.4
.
Time: 0.018

OK (1 test)

hfsd>
```

SUCCESS!

The test compiles now, as does the OrderInformation class.

With this test passing, you're ready to write the next test, still focusing on your first task. That's it—you've just made it through your first round of test-driven development. Remember, the goal was to write just the code you needed to get that test to pass.

Rule #2: Implement the SIMPLEST CODE POSSIBLE to make your tests pass.

Seriously? You made an empty class to get a test to pass and you call that SUCCESS?

Test-driven development is about doing the simplest thing you can to get your test to pass.

Resist the urge to add anything you *might need in the future*. If you need that something later, you'll write a test then and the code to pass that test. In the meantime, leave it alone. Obviously you can't stop here—you need to move on to the next test—but **focusing on small bits of code** is the heart and soul of test-driven development.

This is the YAGNI principle... You Ain't Gonna Need It

Red, green, refactor...

Test-driven development works on a very simple cycle:

1. **Red: Your test fails.**
First you write a test that checks whatever functionality you're about to write. Obviously it fails, since you haven't implemented the functionality yet. This is the **red stage**, since your test GUI probably shows the test in red (failing).

2. **Green: Your test passes.**
Next, implement the functionality to get that test to pass. That's it. No more. Nothing fancy. Write the ***simplest code*** you can to get your test to pass. This is the **green stage**.

3. **Refactor: Clean up any duplication, ugliness, old code, etc.**
Finally, after your test passes, you can go back in and clean up some things that you may have noticed while implementing your code. This is the **refactor stage**. In the example for Starbuzz, you don't have any other code to refactor, so you can go right on to the next test.

When you're done refactoring, move on to the next test and go through the cycle again.

Exercise

Below is the task we're working on and the user story it came from. Your job is to add the next test to the `TestOrderInformation` class to make progress on this task.

You should always look to the story to figure out what you should be testing at a higher, functional level.

For this test you should be focusing on the OrderInformation class. We'll get to the gift card and receipt later.

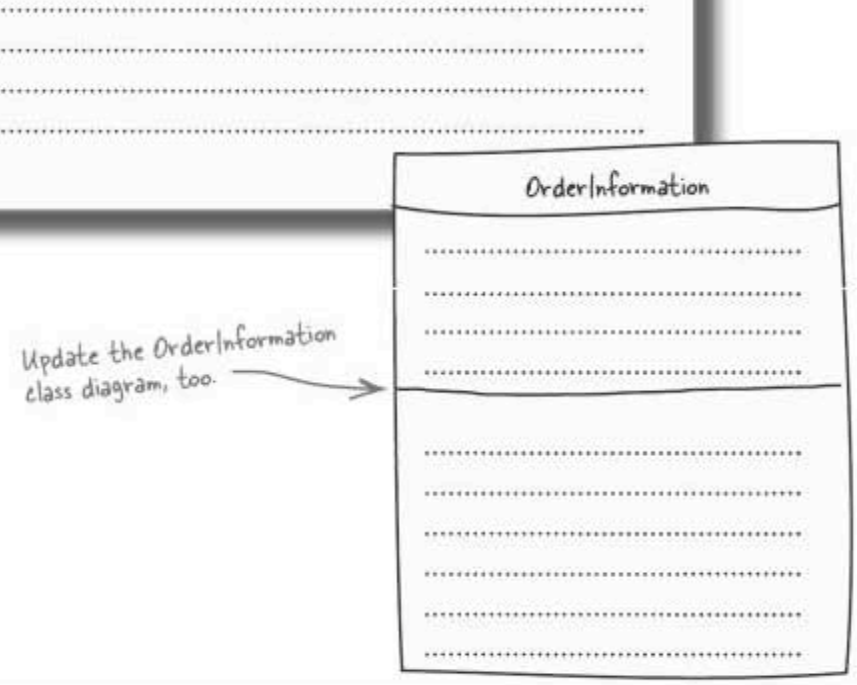
```
import org.junit.*;

public class TestOrderInformation {
  @Test
  public void testCreateOrderInformationInstance() {
    OrderInformation orderInfo = new OrderInformation();
  }

  @Test
  public void testOrderInformation() {
    .....
    .....
    .....
    .....
    .....
  }
}
```

* If you're not a Java programmer, try and write out the test in the framework you're using, or type it into your IDE.

Now implement the code to make your test pass. Remember, you just want the simplest code possible to get the test passing.

Here's the OrderInformation class created to pass the first test. You need to fill it out to pass the test you just wrote.

```
public class OrderInformation {

  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
}
```



Exercise Solution

Below is the task we're working on and the user story it came from. Your job is to add the next test to the `TestOrderInformation` class to make progress on this task.

Title: Preorder your coffee with a gift card

Description: Select your coffee preferences from the options, enter your gift card number, name, preferred store, and click submit to get a confirmation number, remaining balance, and estimated time when it will be ready for pickup.

Priority: 20

Task 1

Capture order info, gift card info, and receipt info.

5

To get the rest of the OrderInformation class together, you need to add coffee preference, gift card number, customer name, and preferred store to the order information.

```
import org.junit.*;

public class TestOrderInformation {
  @Test
  public void testCreateOrderInformationInstance() { // existing test }

  @Test
  public void testOrderInformation() {
    OrderInformation orderInfo = new OrderInfomation();
    orderInfo.setCustomerName("Dan");
    orderInfo.setDrinkDescription("Mocha cappa-latte-with-half-whip-skim-fracino");
    orderInfo.setGiftCardNumber(123456);
    orderInfo.setPreferredStoreNumber(8675309);
    assertEqual(orderInfo.getCustomerName(), "Dan");
    assertEqual(orderInfo.getDrinkDescription(),
      "Mocha cappa-latte-with-half-whip-skim-fracino");
    assertEqual(orderInfo.getGiftCardNumber(), 123456);
    assertEqual(orderInfo.getPreferredStoreNumber(), 8675309);
  }
}
```

Our test simply creates the OrderInformation, sets each value we need to track, and then checks to make sure we get the same values out

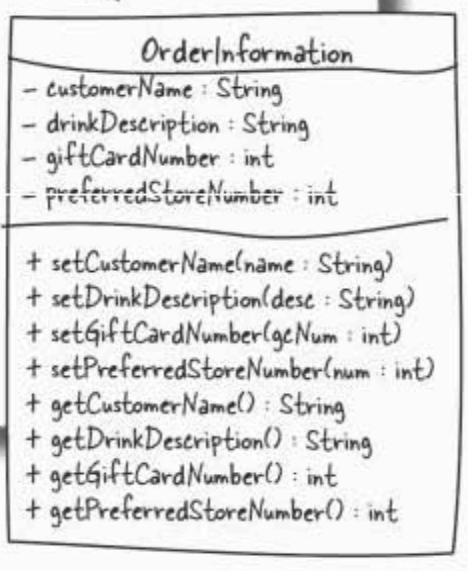
You might want to use constants in your own code, so you don't have any typos beween setting values and checking against the returned values (especially in those long coffee-drink names).

Now implement the code to make your test pass. Remember, you just want the simplest code possible to get the test passing.

```
public class OrderInformation {

  private String customerName;
  private String drinkDescription;
  private int giftCardNumber;
  private int preferredStoreNumber;

  public void setCustomerName(String name) {
    customerName = name;
  }
  public void setDrinkDescription(String desc) {
    drinkDescription = desc;
  }
  public void setGiftCardNumber(int gcNum) {
    giftCardNumber = gcNum;
  }
  public void setPreferredStoreNumber(int num) {
    preferredStoreNumber = num;
  }
  public String getCustomerName() {
    return customerName;
  }
  public String getDrinkDescription() {
    return drinkDescription;
  }
  public int getGiftCardNumber() {
    return giftCardNumber;
  }
  public int getPreferredStoreNumber() {
    return preferredStoreNumber;
  }
}
```

This class is really just a few member variables, and then methods to get and set those variables.

Is there anything less you could do here and still pass the test case?



In TDD, tests DRIVE your implementation

Now you've got a working and tested `OrderInformation` class. And, because of the latest test, you've got getters and setters that all work, too. In fact, the things you put in the class were completely driven by your tests.

Test-driven development is different from just *test-first development* in that it drives your implementation ***all the way through development***. By writing your tests before your code, you have to focus on the functionality right away. What exactly is the code you're about to write actually supposed to do?

To help keep your tests manageable and effective, there are some good habits to get into:

1. **Each test should verify ONLY ONE THING**
   To keep your tests straightforward and focused on what you need to implement, try to make each test only test one thing. In the Starbuzz system, each test is a method on our test class. So `testCreateOrderInformation()` is an example of a test that only checks one thing: all it does is test creating a new order object. The next test, which tests multiple methods, still tests only one piece of functionality: that the order stores the right information within it.

2. **AVOID DUPLICATE test code**
   You should try to avoid duplicated test code just like you'd try to avoid duplicated production code. Some testing frameworks have setup and teardown methods that let you consolidate code common to all your tests, and you should use those liberally. You also may need to mock up test objects—we'll talk more about how to do that later in this chapter.

   Suppose you need a database connection: you could set that up in your setup() method, and release the connection in your teardown() method of your test framework.

3. **Keep your tests in a MIRROR DIRECTORY of your source code**
   Once you start using TDD on your project, you'll write tons of tests. To help keep things organized, keep the tests in a separate directory (usually called `test/`) at the same level as your source directory, and with the same directory structure. This helps avoid problems with languages that assume that directories map to package names (like Java) while keeping your tests cases out of the way of your production code. This also makes things easier on your build files, too; all tests are in one place.

there are no Dumb Questions

Q: If TDD drives my implementation, when do we do design?

A: TDD is usually used with what's called **evolutionary design**. Note that this **doesn't** mean code all you want, and magically you'll end up with a nicely designed system. The critical part of getting to a good design is the refactoring step in TDD. Basically TDD works hard to prevent overdesigning something. As you add functionality to your system, you'll be increasing the code base. After a while you'll see things getting naturally disorganized, so after you get your test to pass, refactor it. Redesign it, apply the appropriate design patterns, whatever it takes. And all along your tests should keep passing and let you know that you haven't broken anything.

Q: What if I need more than one class to implement a piece of functionality?

A: That's fine functionally, but you should really consider adding tests for each class you need to realize the functionality. If you add tests for each class, you'll add a test, implement the code, add a test, etc., and build up your functionality with the red, green, refactor cycle.

Q: The test example we just did had us writing tests for getter and setter methods. I thought we weren't supposed to test those.

A: There's nothing wrong with testing setters and getters; you just don't get much bang for the buck. The setter and getter example was just the beginning. The next few pages really dig into a challenging TDD problem.

Q: So when I implement code to make a particular test pass, I know what the next test I have to write is. Can't I just add the code I'm going to need for that test too?

A: No. There's a couple problems with that approach. First, it's a really slippery slope once you start adding things that are outside of the scope of the test you're trying to get to pass. You might think you need it, but until a test says you do, don't tempt yourself.

The second, and possibly more severe problem is that if you add code now for the next test you're going to write, that second test probably won't fail. Which means you don't know that it's actually testing what you think it is. You can't be sure that it will let you know if the underlying code breaks. Write the test—then implement code for that test.

Test-driven development is all about creating tests for specific functionality, and then writing code to satisfy that functionality.

Anything beyond that functionality is NOT IMPORTANT to your software (right now).

Finish up the remaining work on the current Starbuzz task by writing tests and then the implementation for the gift card and receipt objects.

We've left the answers out on this one...it's up to you to write these tests on your own.

Completing a task means you've got all the tests you need, and they all pass

To finish up the first task, you'll need to be able to test that order, gift card, and receipt information can be captured and accessed. You should have created objects for all three of these items. Here's how we implemented each object...

Task 1

Capture order info, gift card info, and receipt info.

5

Here are the classes that came out of our first task. All of the fields came from data the story said was captured.

OrderInformation

- customerName : String
- drinkDescription : String
- giftCardNumber : int
- preferredStoreNumber : int

+ setCustomerName(name : String)
+ setDrinkDescription(desc : String)
+ setGiftCardNumber(gcNum : int)
+ setPreferredStoreNumber(num : int)
+ getCustomerName() : String
+ getDrinkDescription() : String
+ getGiftCardNumber() : int
+ getPreferredStoreNumber() : int

Receipt

- confirmationNumber : int
- pickupTime : Date
- gcBalance : BigDecimal

+ setConfirmationNumber(no : int)
+ setPickupTime(date : Date)
+ setGCBalance(bal : BigDecimal)
+ getConfirmationNumber() : int
+ getPickupTime() : Date
+ getGCBalance() : BigDecimal

GiftCard

- activationDate : Date
- expirationDate : Date
- balance : BigDecimal

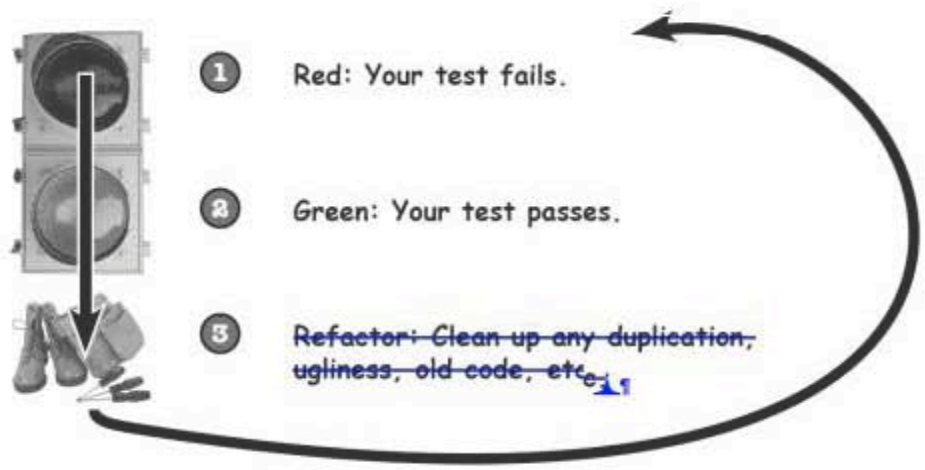
+ getActivationDate() : Date
+ getExpirationDate() : Date
+ getBalance() : BigDecimal
+ setBalance(BigDecimal) : void
+ equals(object : Object) : boolean

When your tests pass, move on!

The first task is complete and we have `Receipt`, `GiftCard`, and `OrderInformation` classes written and tested. Now it's time to try our TDD approach on a tougher task: implementing the business logic to process and store orders.

Different task, same process

This task is no different than the last one. We'll just follow the same approach. Write a test that fails, implement the code to get the test passing, perform any cleanup, and then repeat.



Red: write (failing) tests

The first step is to write a test. The user story says we need to process and store order information, so let's assume we'll need a new class for that, called `OrderProcessor`:

```
import org.junit.*;

public class TestOrderProcessor {
  @Test
  public void testCreateOrderProcessor() {
    OrderProcessor orderProcessor = new OrderProcessor();
  }
}
```

There's nothing special about the name OrderProcessor. It's just a place to put business logic, since the only other classes in the app are for storing data.

```
File Edit Window Help Failure
hfsd> javac -cp junit.jar
       headfirst.sd.chapter8.TestOrderProcessor.java

TestOrderProcessor.java:8: cannot find symbol
symbol  : class OrderProcessor
location: class headfirst.sd.chapter8.TestOrderProcessor
    OrderProcessor orderProcessor = new OrderProcessor();
    ^
TestOrderProcessor.java:8: cannot find symbol
symbol  : class OrderProcessor
location: class headfirst.sd.chapter8.TestOrderProcessor
    OrderProcessor orderProcessor = new OrderProcessor();
                                    ^
2 errors

hfsd>
```

As you would expect, this test will fail—you don't have an `OrderProcessor` yet. So now you can fix that pretty easily.

This test doesn't even compile, let alone pass.

Green: write code to pass tests

To get your first test to pass, just add an empty `OrderProcessor` class:

```
public class OrderProcessor {
}
```

Green: test compiles and passes.

```
File Edit Window Help Success
hfsd> javac -d bin -cp junit.jar *.java


hfsd> java -cp junit.jar;.\bin org.junit.runner.
JUnitCore headfirst.sd.chapter8.TestOrderProcessor
JUnit version 4.4

Time: 0.018

OK (1 test)

hfsd>
```

That's it. Recompile, retest, and you're back to green. The user story says you need to process and store order information. You've already got classes that represent order information (and a receipt), so use those now along with the `OrderProcessor` class that you just created.

Red

Below is a new test method. Implement a test that will verify your software can process a simple order.

You'll need to put the pieces together to describe the order...

...and then pass it on to the order processor and make sure it worked.

```
// other tests

@Test
public void testSimpleOrder() {
  OrderProcessor orderProcessor = new OrderProcessor();
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
}
```

Don't forget about your classes from the last task.

Red

Your job was to implement a test that will verify your software can process a simple order.

You can just make up a gift card number here...

The simplest thing here is to not worry about the balance on the card... this is just testing the simplest version of order processing.

```
// existing tests

@Test
public void testSimpleOrder() {

    // First create the order processor
    OrderProcessor orderProcessor = new OrderProcessor();

    // Then you need to describe the order that should be placed
    OrderInformation orderInfo = new OrderInformation();
    orderInfo.setCustomerName("Dan");
    orderInfo.setDrinkDescription("Bold with room");
    orderInfo.setGiftCardNumber(12345);
    orderInfo.setPreferredStoreNumber(123);

    // Hand the order off to the order processor and check the receipt
    Receipt receipt = orderProcessor.processOrder(orderInfo);
    assertNotNull(receipt.getPickupTime());
    assertTrue(receipt.getConfirmationNumber() > 0);
    assertTrue(receipt.getGCBalance().equals(0));

}
```

there are no Dumb Questions

Q: How can you just assume that the gift card has the right amount on it? Isn't that an assumption? Aren't those bad?

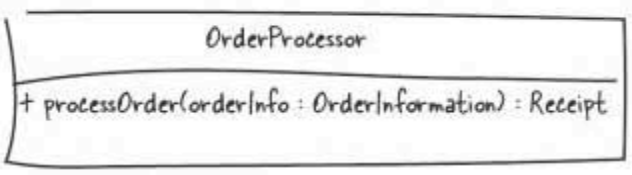
A: We're writing our first test, and then we need to make it pass. So, we're sort of assuming that the gift card has enough on it, but since we're about to implement the backend code, we can make sure it does then. What we are setting ourselves up for is some refactoring. Once we get this test passing we'll obviously need to add a test for a gift card that doesn't have enough money on it. When we do that, we'll certainly have to revisit the code we wrote to get this test going and rework it to support different gift cards and different values. But, this is going to take some thought. Read on...

Q: There are a bunch of values in that test that aren't constants—should I care?

A: Yes, you should. To keep the code sample short we didn't pull those values into constants, but you should treat your test code just like production code you write and apply the same style and discipline. Remember, this isn't throwaway code; it lives in the repository with the rest of your system, and you rely on it to let you know if things aren't working right. Treat it with respect.

Simplicity means avoiding dependencies

Let's add a `processOrder()` method to `OrderProcessor`, since that's what our latest test needs to pass. The method should return a `Receipt` object, like this:



But here's where things get tricky: `processOrder()` needs to connect to the Starbuzz database. Here's the task that involves that piece of the system's functionality:

Wait a second...what happened to the simplest code possible? Can't we just simulate a database, and save writing the actual database code for when we get to the later task?

Dependencies make your code more complex, but the point of TDD is to keep things as simple as possible.

You've got to have `processOrder()` talk to a database, but the database access code is part of another task you haven't dealt with yet.

On top of that, is the simplest code possible to get this test to pass really to write database-access code?

What would you do in this situation?

Always write testable code

When you first start practicing TDD, you will often find yourself in situations where the code you want to test seems to depend on everything else in your project. This can often be a maintenance problem later on, but it's a huge problem ***right now*** when it comes to TDD. Remember our rules? We really don't want that "simplest thing" to be "an order processor with a database connection, four tables, and a full-time DBA."

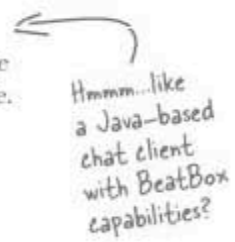
Rule #2: Implement the SIMPLEST CODE POSSIBLE to make your tests pass.

All real-world code has dependencies

When you only have basic classes in your system, it's not too hard to split things up so you can test pieces one thing at a time. But eventually, you're going to have code that depends on something external to your system, like a database.

This can show up lots of other ways too, though: your system might depend on a network connection to send or receive data, or you might need to read data from files that are created by another application, or you might need to use a sound API to generate annoying thumps and beeps. In all of these cases, the dependencies make it hard to test one thing at a time.

But that doesn't mean you don't have to test. It just means you have to figure out a way to test things ***independent*** of all those dependencies.

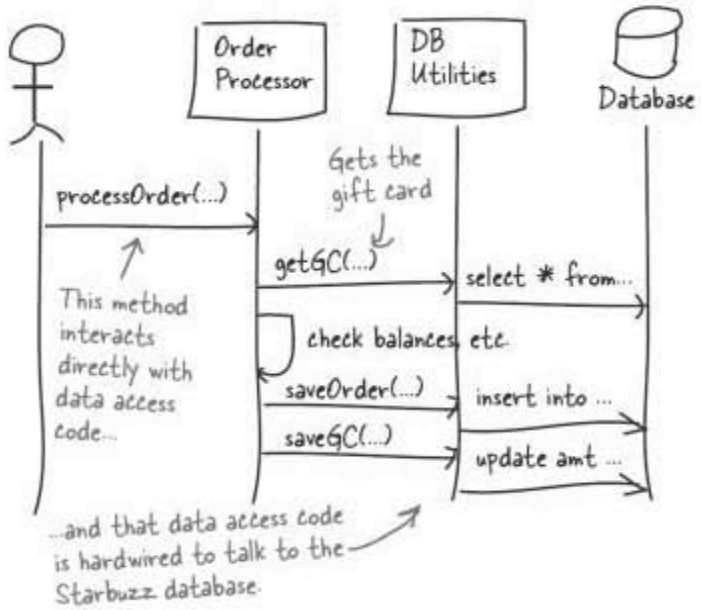


When things get hard to test, examine your design

One of the first things you can do to remove dependencies is to see if you can remove the dependencies. Take a look at your design, and see if you really need everything to be as ***tightly coupled***—or interdependent—as your current design calls for. In the case of Starbuzz, here's what we've assumed so far:

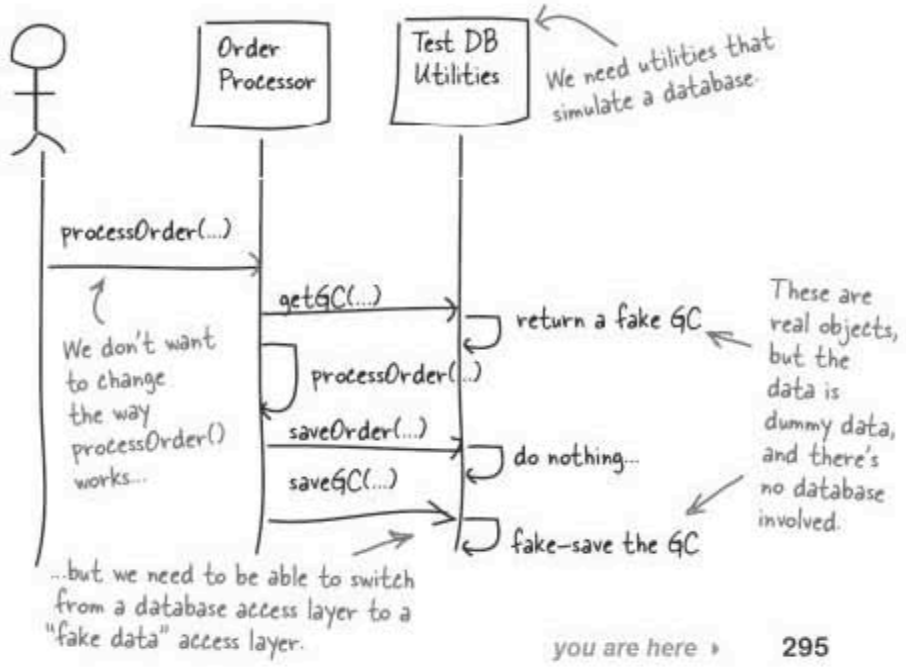
What we have...

The order processor has to fetch gift cards from the database, check the order, save it, and update the gift card (again in the database). So `processOrder()` is hardwired to connect to the database...and that's what makes testing the method tricky.



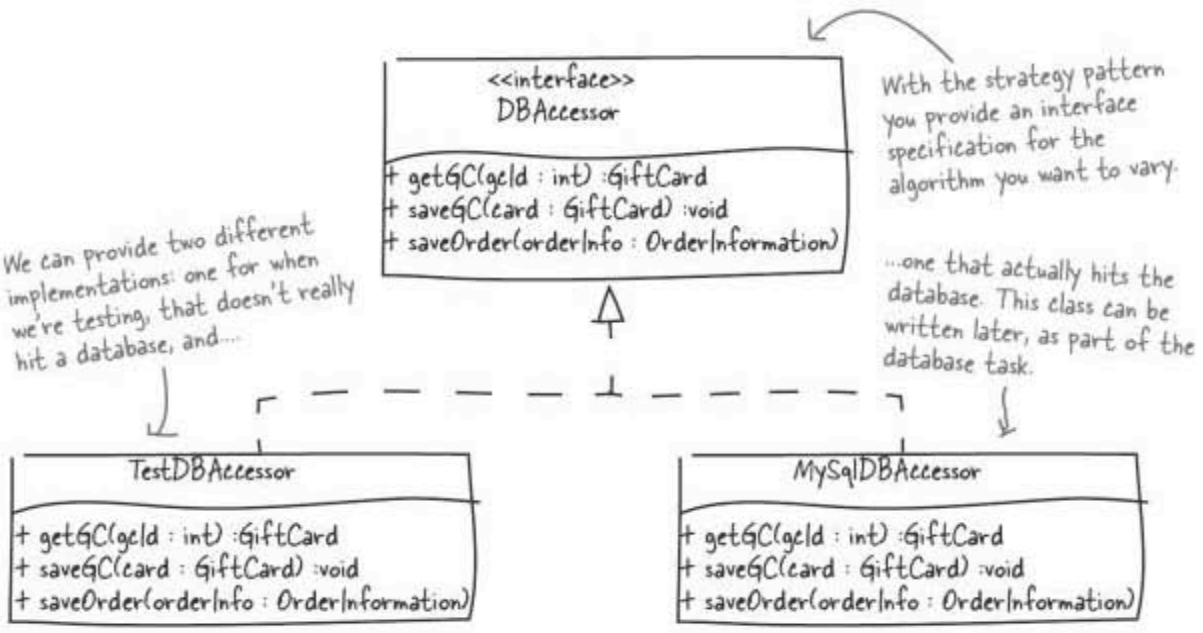
What we need...

How can we have `processOrder()` make the same calls, but avoid database access code? We need a way to get data ***without*** requiring a database—it's almost like we need a fake data access layer.



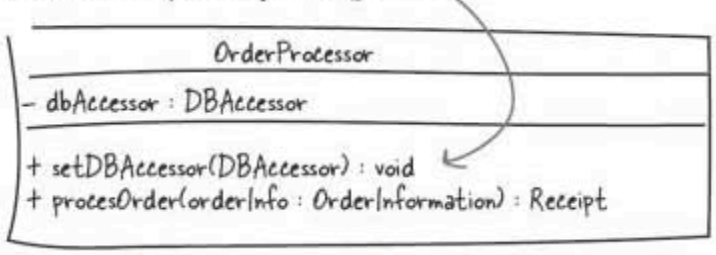
The strategy pattern provides for multiple implementations of a single interface

We want to hide how the system gets gift cards, and vary it depending on whether we're testing the code or we're running the system in production. Flip to Chapter 1 of *Head First Design Patterns* and you'll find there's a ready made pattern to help us deal with just this problem: the **strategy pattern**.



Now we've got two *different* ways of hitting the database, and `OrderProcessor` doesn't need to know which one it's using. Instead, it just talks to the `DBAccessor` interface, which hides the details about which implementation is actually used.

All we need to do now is add a way to give the OrderProcessor the correct DBAccessor implementation, based on whether the test code or the system is providing it.



* If your customer was unsure about what database they might use in production, this same approach would make it easy to swap out database vendors and implementations.

The strategy pattern encapsulates a family of algorithms and makes them interchangeable.

Getting to Green... again

Now you've got a way to isolate the OrderProcessor class from the database. Implement the processOrder() method using the right database strategy.

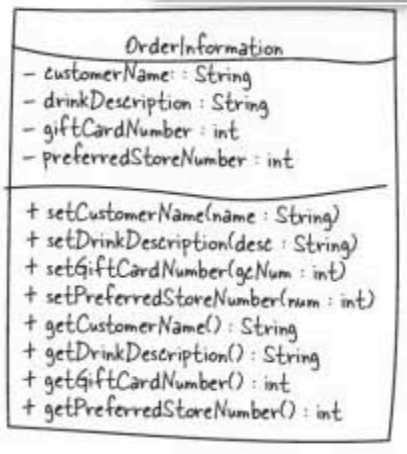
You'll need to pull the gift card from the database

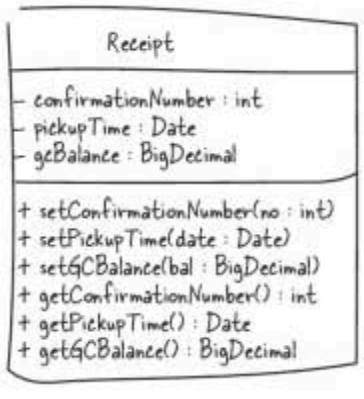
...save the order...

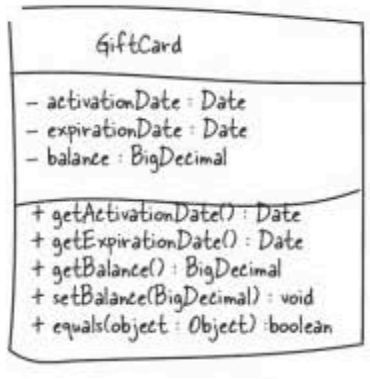
...then save the updated gift card back out

This allows the right database accessor to be set for order processing.

```
// existing code

private DBAccessor mDBAccessor;
public void setDBAccessor(DBAccessor accessor) {
  dbAccessor = accessor;
}
public Receipt processOrder(OrderInformation orderInfo) {

  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....
  .....

  }
}
```







Getting to Green...again.

Now you've got a way to isolate the OrderProcessor class from the database. Implement the processOrder() method using the right database strategy.

Remember, as long as you're using the test DBAccessor this is just a placeholder.

The test wants a zero-balance gift card at the end. So we simulate that.

Hmm, this isn't good; this is what the test wants but we're obviously going to have to revisit this. We'll need another test.

```
// existing code

private DBAccessor dbAccessor;
public void setDBAccessor(DBAccessor accessor) {
  mDBAccessor = accessor;
}
public Receipt processOrder(OrderInformation orderInfo) {

  GiftCard gc = dbAccessor.getGC(orderInfo.getGiftCardNumber());
  dbAccessor.saveOrder(orderInfo);

  // This is what our test is expecting
  gc.setBalance(new BigDecimal(0));

  dbAccessor.saveGC(gc);

  Receipt receipt = new Receipt();
  receipt.setConfirmationNumber(12345);
  receipt.setPickupTime(new Date());
  receipt.setGCBalance(gc.getBalance());

  return receipt;
  }
}
```

Remember, this is just the code needed to get our test passing; it's OK that we're going to have to revisit this code for the next test.

there are no Dumb Questions

Q: I just don't buy it. We just wrote a bunch of code that we know is wrong. How is this helping me?

A: The test we wrote is valid—we need that test to work. The code we wrote makes that test work so we can move on to the next one. That's the principle behind TDD—just like we broke stories into tasks to get small pieces, we're breaking our functionality into small code pieces. It didn't take long to write the code to get the first test to pass and it won't take long to refactor it to get the second one to pass, or the third. When you're finished you'll have a set of tests that makes sure the system does what it needs to, and you won't have any more code than necessary to do it.

Keep your test code with your tests

All that's left is to write up an implementation of `DBAccessor` for the `processOrder()` method to use, and finish the `testSimpleOrder()` test method. But the test implementation of `DBAccessor` is really only used for tests, so it belongs with your testing classes, ***not*** in your production code:

All this code is in our test class, which is in a separate directory from production code.

```
public class TestOrderProcessing {
  // other tests
```

Here's a simple DBAccessor implementation that returns the values we want.

```
public class TestAccessor implements DBAccessor {
  public GiftCard getGC(int gcId) {
    GiftCard gc = new GiftCard();
    gc.setActivationDate(new Date());
    gc.setExpirationDate(new Date());
    gc.setBalance(new BigDecimal(100));
  }
  // ... the other DBAccessor methods go here...
}
```

Since this is only used for testing, it's defined inside our test class.

```
  @Test
  public void testSimpleOrder() {
    // First create the order processor
    OrderProcessor orderProcessor = new OrderProcessor();
    orderProcessor.setDBAccessor(new TestAccessor());
```

Set the OrderProcessor object to use the test implementation for database access—which means no real database access at all.

```
    // Then we need to describe the order we're about to place
    OrderInformation orderInfo = new OrderInformation();
    orderInfo.setCustomerName("Dan");
    orderInfo.setDrinkDescription("Bold with room");
    orderInfo.setGiftCardNumber(12345);
    orderInfo.setPreferredStoreNumber(123);

    // Hand it off to the order processor and check the receipt
    Receipt receipt = orderProcessor.processOrder(orderInfo);
    assertNotNull(receipt.getPickupTime());
    assertTrue(receipt.getConfirmationNumber() > 0);
    assertTrue(receipt.getGCBalance().equals(0));
  }
}
```

With the testing database accessor, we can test this method, even without hitting a live database.

Remember, this was all about the simplest code possible to return the expected values here.

Testing produces better code

We've been working on testing, but writing tests first has done more than just test our system. It's caused us to organize code better, keeping production code in one place, and everything else in another. We've also written simpler code—and although not everything in the system works yet, the parts that do are streamlined, without anything that's not absolutely required.

And, because of the tight coupling between our system's business logic and database code, we implemented a design pattern, the strategy pattern. Not only does this make testing easier, it decouples our code, and even makes it easy to work with different types of databases.

So testing first has gotten us a lot of things:

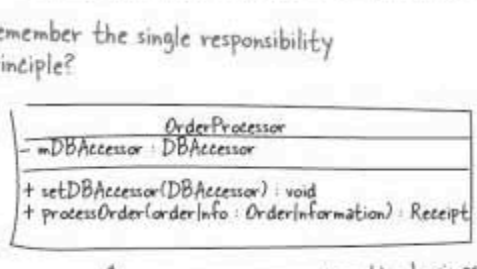
- ☑ **Well-organized code.** Production code is in one place; testing code is in another. Even implementations of our database access code used for testing are separate from production code.
- ☑ **Code that always does the same thing.** Lots of approaches to testing result in code that does one thing in testing, but another in production (ever seen an `if (debug)` statement?). TDD means writing production code, all the time.

Our test uses a testing-specific implementation of DBAccessor, but the order processor runs the same code, because of our strategy pattern, in testing or in production.

- ☑ **Loosely coupled code.** Tightly coupled systems are brittle and difficult to maintain, not to mention really, really hard to test. Because we wanted to test our code, we ended up breaking our design into a loosely coupled, more flexible system.

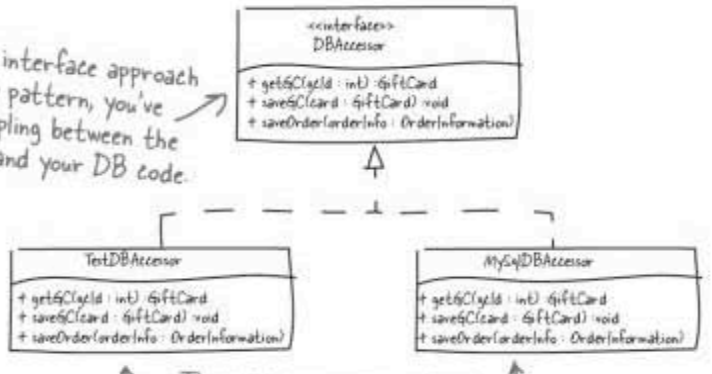
Ever heard your computer science professor or lead architect talking about low coupling and high cohesion? This is what they were talking about. We have low coupling because of our use of interfaces and the strategy pattern, and we've got high cohesion by having our database and business logic code concentrated into separate but well defined classes.

Remember the single responsibility principle?



OrderProcessor has the business logic to handle an order, and doesn't worry about databases. So it's got high cohesion.

Because of the interface approach of the strategy pattern, you've reduced the coupling between the OrderProcessor and your DB code.



These accessors worry about database access, and only database access. That's high cohesion.

Are you kidding me? Did you look at that code we just wrote? We never once look at the expiration date on a gift card, and we always set the balance to 0. How can you call this **better code**?

Your code may be incomplete, but it's still in better shape.

Remember the second rule of test-driven development?

Rule #2: Implement the SIMPLEST CODE POSSIBLE to make your tests pass.

Even though not everything works, the code that we do have works, is testable, and is slim and uncluttered. However, it's pretty clear that we still have lots of work left. The goal is getting everything else working and keeping any additional code just as high-quality as what you've got so far.

All of this...testing for functionality, edge cases, and hokey implementations, adds up to a complete testing approach.

So once you get your basic tests, start thinking about what else you need to test...which will motivate the next piece of functionality to write code for. Sometimes it's obvious what to test next, like adding a test to deal with gift card balances. Other times, the user story might detail additional functionality to work on. And once all that's done, think about things like testing for boundary conditions, passing in invalid values, scalability tests, etc.

BRAIN POWER

We've implemented the basic success-case test for processing an order, but there are clearly problems with our implementation. Write another test that finds one of those problems, and then write code to get the test to pass.

More tests always means lots more code

The gift card class for Starbuzz has four attributes, so we're going to need several tests to exercise those attributes. We could test for:

- ☐ A gift card with more than enough to cover the cost of the order
- ☐ A gift card without enough to cover the cost of the order
- ☐ An invalid gift card number
- ☐ A gift card with exactly the right amount
- ☐ A gift card that hasn't been activated
- ☐ A gift card that's expired

In each case, we need a gift card object with a slightly different set of values, so we can test each variation in our order processing class.

And that's just for the gift cards. You'll need tests for variations on the `OrderInformation` class, too...and we still haven't tested for the bigger failure cases, like what happens if the database fails to save an order.

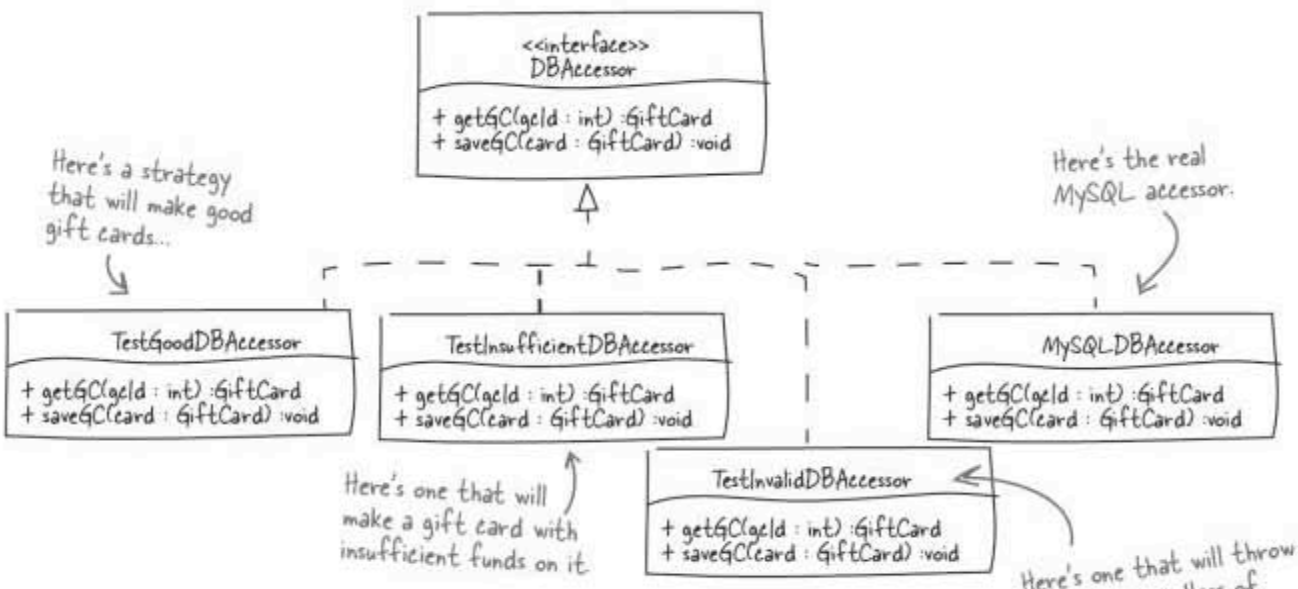
Automated test-driven development means a LOT of test code.

The more functionality you have, the more tests you'll need. And more tests means more code... lots and lots of code. But all that code also means a lot more stability. You'll know your system is working, at every step of the way.

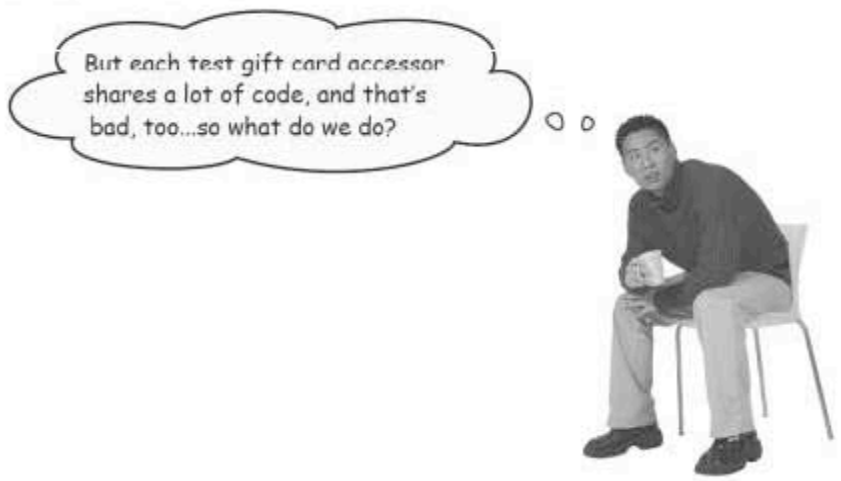
And sometimes, you may not need quite as much code as you first thought...

Strategy patterns, loose couplings, object stand ins...

Suppose we used the strategy pattern again for all the different variations on the types of gift card a database could return, like this:

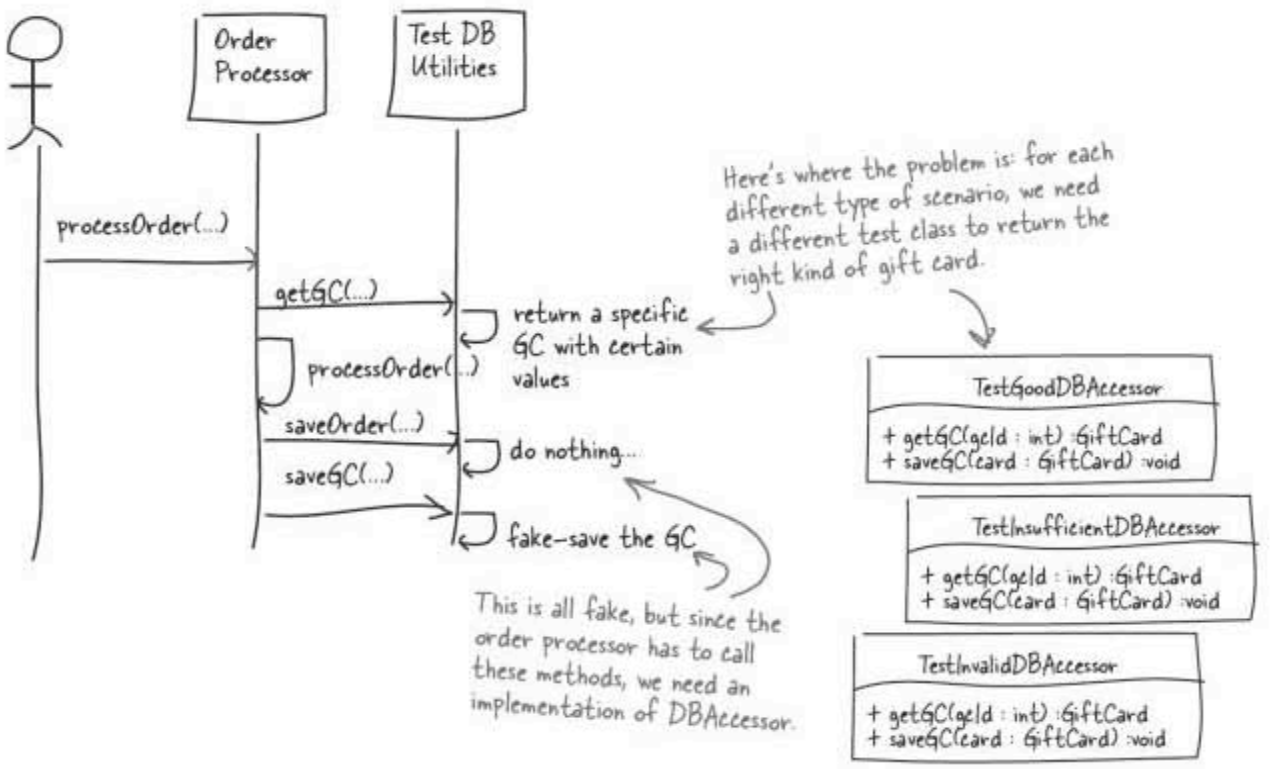


To avoid all these extra classes, you *could* have one `TestDBAccessor` implementation that returned different cards based on the ID you gave it, but that's screwing up loose coupling. `TestDBAccessor` would have to be in sync with your test code to make sure they agree on what each ID means.



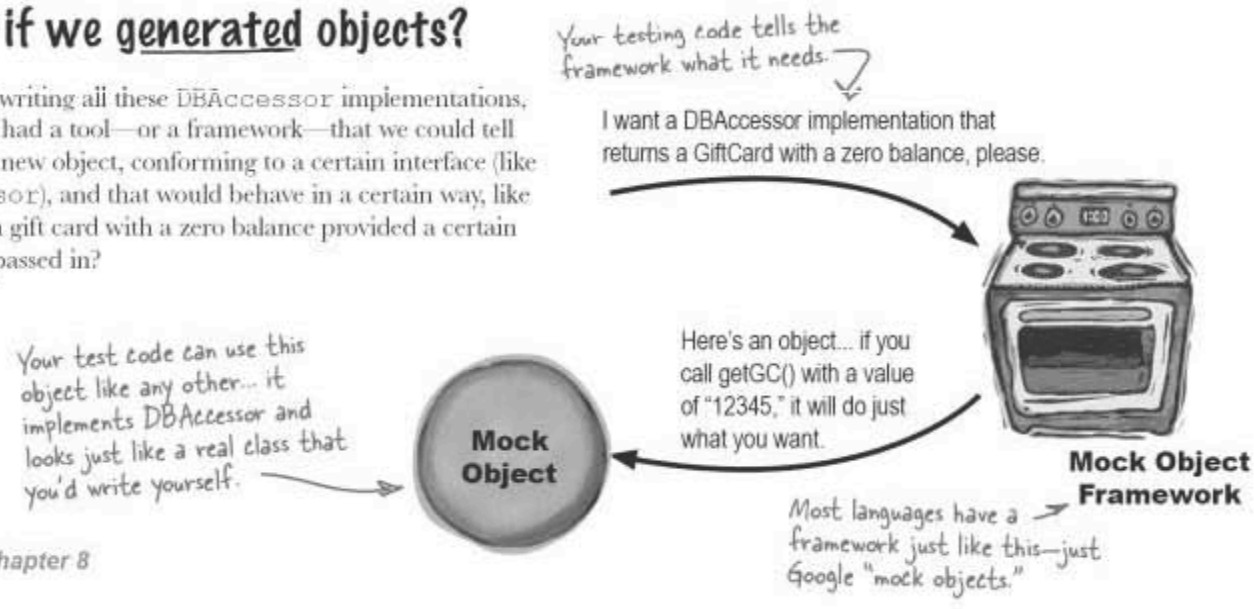
We need lots of different, but similar, objects

The problem right now is that we have a sequence like this:



What if we generated objects?

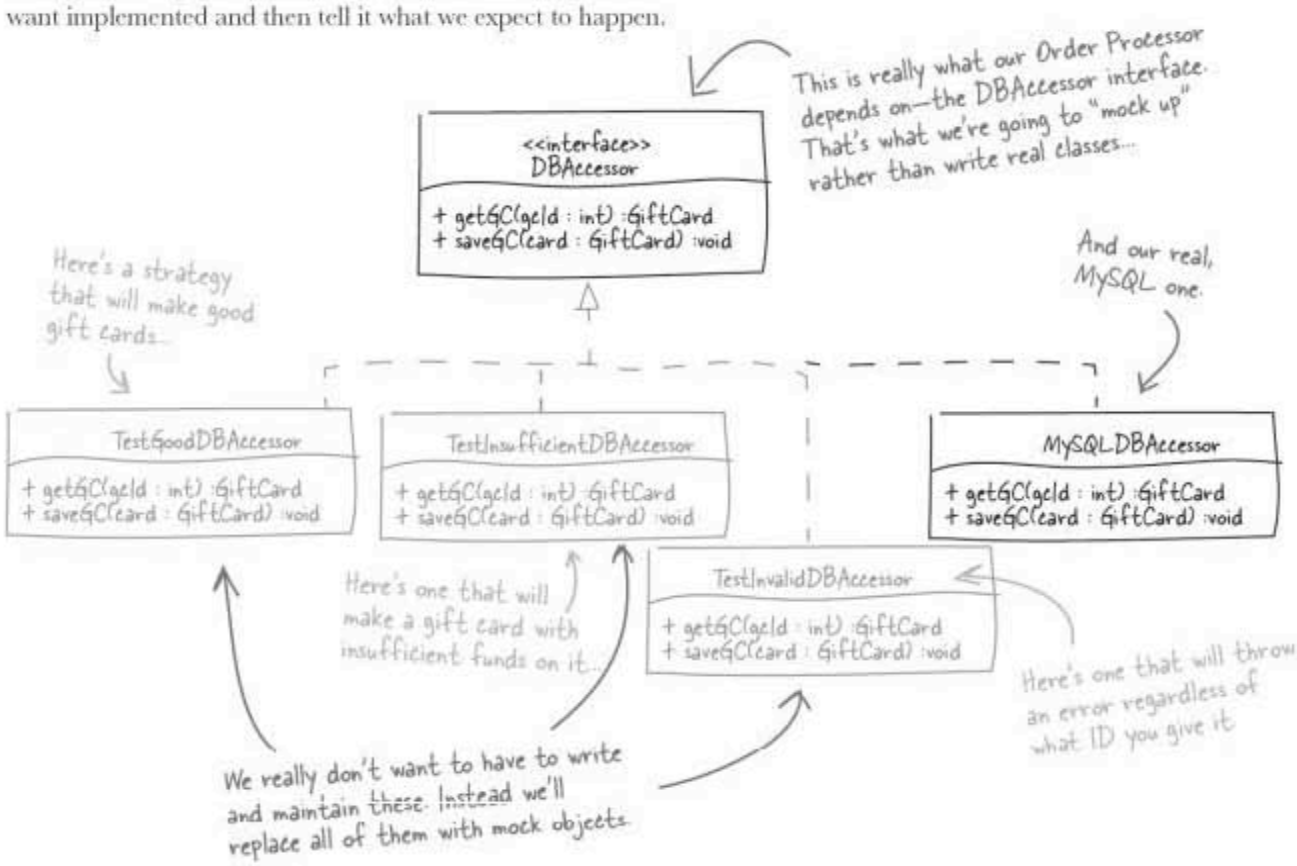
Instead of writing all these `DBAccessor` implementations, what if we had a tool—or a framework—that we could tell to create a new object, conforming to a certain interface (like `DBAccessor`), and that would behave in a certain way, like returning a gift card with a zero balance provided a certain input was passed in?



A mock object stands in for real objects

There's really no need for three different accessors, all of which create a new `GiftCard` object and populate it with different data. That's a lot of extra code to instantiate a `GiftCard` and call some setter methods.

Since we have an interface that describes what each of these implementations should look like, we can take advantage of a **mock object framework** to do the heavy lifting. Instead of implementing all of the classes ourselves, we can give the framework the interface we want implemented and then tell it what we expect to happen.



The mock framework will handle creating implementations of the interface and keeping track of what methods we say should be called, what they should return when they are called, what shouldn't be called, etc. The mock framework's implementation of our interface will track all of this and throw an error if something doesn't go according to the plan we gave it.

* We're going to use the EasyMock framework here but a mock object framework exists for most languages and they all work similarly.

Mock objects are working object stand-ins

Let's look at a mock object framework in action. Below is a test that uses the EasyMock framework, a mock object framework for Java. A good mock object framework allows you to **simulate an object's behavior**, without writing code for that object.

```
import org.easymock.*;
// This test will test placing an order with a valid gift card
// with exactly the right amount of money on it.
@Test
public void testSimpleOrder() {
  // Set everything up and get ready
  OrderInformation orderInfo = new OrderInformation();
  orderInfo.setCustomerName("Dan");
  orderInfo.setDrinkDescription("Bold with room");
  orderInfo.setGiftCardNumber(12345);
  orderInfo.setPreferredStoreNumber(123);
  Date activationDate = new Date(); // Valid starting today
  Date exipirationDate = new Date(activationDate.getTime() + 3600);
  BigDecimal gcValue = new BigDecimal("2.75"); // Exactly enough
  GiftCard startGC =
    new GiftCard(activationDate, expirationDate, gcValue);
  BigDecimal gcEndValue = new BigDecimal("0"); // Nothing left
  GiftCard endGC =
    new GiftCard(activationDate, expirationDate, gcEndValue);

  // Here's where the mock object creation happens
  DBAccessor mockAccessor = EasyMock.createMock(DBAccessor.class);
```

Whatever framework you use, you'll need to import the right classes.

This is all part of the test orderInfo object we want to use.

This is all "normal" test code... no mock objects involved yet

This sets up test values that we'll use in the GiftCard we're testing.

We need a gift card representing the starting values we're testing...

...and then an "ending" gift card. This has what should be returned from testing order processing.

We want an object that implements this interface...

...so we tell our framework to create a mock object that implements the right interface.

At this point, the mock object framework doesn't know much—just that it has to create a stand-in for the DBAccessor class. So it knows the methods it "mocks", but nothing more than that—no behavior yet at all.

Once you create a mock object, it's in "record mode." That means you tell it what to expect and what to do...so when you put it in replay mode, and your tests use it, you've set up exactly what the mock object should do.

Remember, you haven't had to write your own class... that's the big win here.

```
// Tell our test framework what to call, and what to expect
EasyMock.expect(mockAccessor.getGC(12345)).andReturn(startGC);

// Simulate processing an order
mockAccessor.saveOrder(orderInfo);

// Then the processor should call saveGC(...) with an empty GC
mockAccessor.saveGC(endGC);

// And nothing else should get called on our mock.
EasyMock.replay(mockAccessor);

// Create an OrderProcessor...
OrderProcessor processor = new OrderProcessor();
processor.setDBAccessor(mockAccessor);
Receipt rpt = processor.processOrder(orderInfo);

// Validate receipt...
}
```

First, expect a call to getGC() with the value 12345... that matches up with the orderInfo object we created over here.

When getGC() is called with that value, return the startGC object... this simulates getting a card from the database, and we've supplied the exact values we want for this test scenario.

This doesn't do anything...but it tells the mock object that you should have saveOrder() called, with orderInfo as the parameter. Otherwise, something's gone wrong, and it should throw an exception.

Then, the mock object should have saveGC() called on it, with the endGC gift card simulating the right amount of money being spent. If this isn't called, with these values, then the test should fail.

Calling replay() tells the mock object framework "OK, something is going to replay these activities, so get ready."

This is like activating the object; it's ready to be used now.

And here's where we use the mock object as a stand-in for a DBAccessor implementation: we test order processing, never having to write a custom implementation for this particular test case (or for any of the other specific test cases we need to check out).

This might seem like a good bit of work here, but we've saved one class. Add in all the other variations of testing for specific gift card things, and you'll save lots of classes...and that's a big deal.

there are no Dumb Questions

Q: These mock objects don't seem to be doing anything I couldn't do myself. What are they buying me again?

A: Mock objects give you a way to create custom implementations of interfaces without needing to actually write the code. Just look at page 303. We needed three different variations of gift cards (if you count the testInvalidGiftCard one). Two of them had different behavior, not just different values. Without the mock objects we'd have to implement that code ourselves. You could do it, but why?

Q: Why didn't we use mock objects for the gift cards themselves?

A: Well, two reasons. First, we'd have to introduce an interface for the gift cards. Since we don't have any behavioral variations it really doesn't make a lot of sense to put an interface here. Second, all we're really changing are the values it returns since it's pretty much a simple data object anyway. We can get that same result by just instantiating a couple different gift cards at the beginning of our test and set them to have the values we want. Mock objects (and the required interface) would be overkill here.

Q: Speaking of interfaces, doesn't this mean I'll need an interface at any point I'd want a mock object in my tests?

A: Yes—and truthfully sometimes you end up putting interfaces in places that you really don't ever intend on having more than one implementation. It's not ideal, but as long as you're aware that you're adding the interface strictly for testing it's not usually a big deal. Generally the value you get from being able to unit-test effectively with less test code makes it worth the trade-off.

Q: What's that replay(...) method all about?

A: That's how you tell the mock object that you're done telling it what's about to happen. Once you call replay on the mock object it will verify any method calls it gets after that. If it gets calls it wasn't expecting, in a different order, or with different arguments, it will throw an exception (and thereby fail your test).

Q: What about arguments...you say they're compared with Java's equals( ) method?

A: Right—EasyMock tests the arguments the mock object gets during execution against the ones you said it should get by using the `equals()` method. This means you need to provide an `equals()` method on classes you use for arguments to methods. There are other comparison operators to help you deal with things like arrays where the reference value is actually compared. Check out the EasyMock docs (*www.easymock.org/Documentation*) for more details.

Q: So we changed our design a pretty good bit to get all this testing stuff going. The design feels.. upside-down. We're telling the OrderProcessor how to talk to the database now...

A: Yes, we are. This pattern is called **dependency injection**, and it shows up in a lot of frameworks. Specifically the Spring Framework is built on the concepts of dependency injection and inversion of control. In general, dependency injection really supports testing—particularly in cases where you need to hide something ugly like a database or the network. It's all about dependency management and limiting how much of the system you need to be concerned about for any given test.

Q: So do you need dependency injection to do good testing or mock objects?

A: No. You could do a lot of what we did with the `DBAccessor` by using a factory pattern that can create different kinds of `DBAccessors`. However, some people feel that dependency injection just feels cleaner. It does have an impact on your design, and it does often mean adding an interface where you might not have put one before, but those typically aren't the parts of your design that cause problems; it's usually that part of the code that no one bothered to look at because time was getting tight and the project had to ship.

Good software is testable...

There are lots of things to think about when designing software: reusability, clean APIs, design patterns, etc. Equally important is to think about your code's testability. We've talked about a few measures of testability like well-factored code and code coverage. However, don't forget that just because you have JUnit running on every commit that your code isn't guaranteed to be good.

There are a few testing bad habits you need to watch out for:

A whole-lotta-nuthin'

If you're new to test driven development it's very easy to write a whole lot of test code but not really test anything. For example, you could write a test that places a Starbuzz order but never checks the gift card value or receipt after the order is placed "Didn't throw an exception? Good to go." That's a lot like saying "it compiles—ship it."

It's still me...

In an overeager attempt to validate data it's easy to go crazy testing fake data you fed into the system initially and miss the actual code you need to test. For example, suppose you write a test that checks that the gift card value and expiration date are correct when you call `getGC()` ... on our `TestDBAccessor`. This is a simplistic example but if you're traversing a few layers of code with your test, it's not too hard to forget that you put the value you're about to test in there in the first place.

Ghosts from the past

You need to be extremely careful that your system is in a known state every time your automated tests kick off. If you don't have an established pattern for how to write your tests (like rolling back database transactions at the end of each test) it's very easy to leave scraps of test results laying around in the system. Even worse is writing other tests that **rely** on these scraps being there. For example, imagine if our end-to-end testing placed an order, and then a subsequent test used the same gift card to test the "insufficient funds" test. What happens the second time this pair of tests execute? What if someone just reruns the second test? Each test should execute from a known, restorable state.

There are a lot of ways to write bad tests—these are just a few of them. Pick your search engine of choice and do a search for "TDD antipatterns" to find a whole lot more. Don't let the possibility of bad tests scare you off, though—just like everything else, the more tests you write the better you'll get at it!

It's not easy bein' green...

You did it—through the help of the strategy pattern, dependency injection (see the previous No Dumb Questions), and mock objects, you have a really powerful, but not too bulky, suite of unit tests. You now have piles of tests that make sure your system does what it's supposed to be doing at all times. So to keep your system in line:

1. Always write a test before you write the real production code.
2. Make sure your test fails, and then implement the simplest thing that will make that test pass.
3. Each test should really only test one thing; that might mean more than one assertion, but one real concept.
4. Once your're back to green (your test passes) you can refactor some surrounding code if you saw something you didn't like. No new functionality—just cleanup and reorganization.
5. Start over with the next test. When you're out of tests to write, you're done!

When all of your tests pass, you're done

Before we never really had a way of knowing when we were finished. You wrote a bunch of code, probably ran it a few times to make sure it seemed to be working, and then moved on. Unless someone said something bad happened, most developers won't look back. With test-driven development we know exactly when we're done—and exactly what works.

Which do you think is better?

Bob again...

Vs

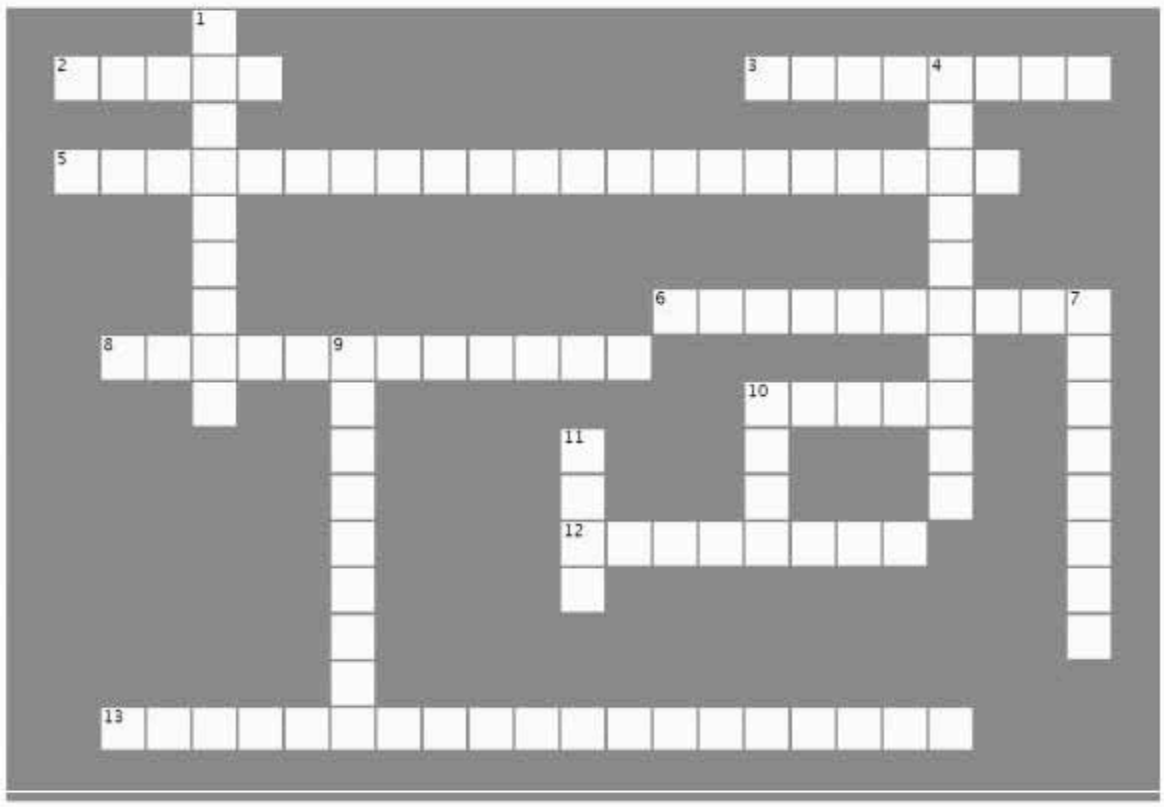
```
File Edit Window Help Bliss
hfsd> java -cp junit.jar;. org.junit.runner.JUnitCore
                   headfirst.sd.chapter8.TestOrderProcessor
JUnit version 4.4
. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . .
. . . . .
Time: 0.321
OK (86 test)

hfsd>
```

Our test suite in action

* If you said Bob, please donate this book to your library.

TDDcross

The crossword tests are below; fill in the answers to make each one pass.



Across

2. You ain't gonna need it.
3. Red, Green, ...
5. TDD.
6. Mock objects realize .....
8. Bad approaches to TDD are called .....
10. TDD means writing tests .....
12. To do effective TDD you need to have low .....
13. To help reduce dependencies to real classes you can use ....

Down

1. Fine grained tests.
4. When you should test.
7. Write the ..... code that will get the test to pass.
9. ..... testing is essential to TDD.
10. Your tests should ..... at first.
11. To help reduce test code you can use ..... objects.

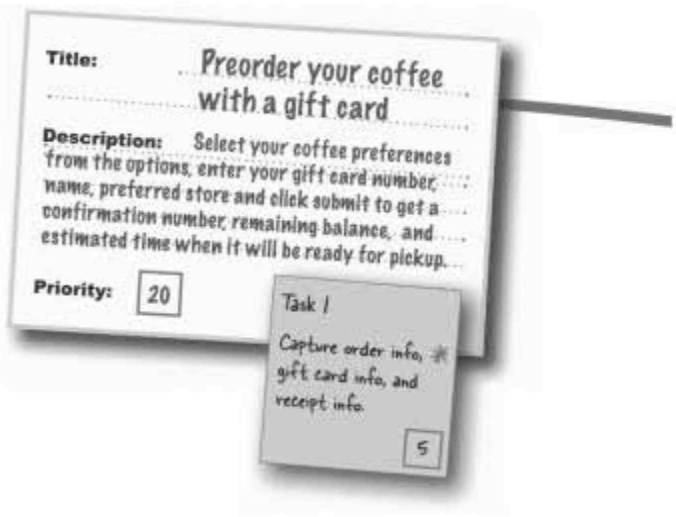
→ Answers on page 315.

A day in the life of a test-driven developer...

Once you have your tests passing, you know you built what you set out to. You're done. Check the code in, knowing that your version control tool will ping your CI tool, which will diligently check out your new code, build it, and run your tests. All night. All the next day. Even when Bob checks in some code that breaks yours...*

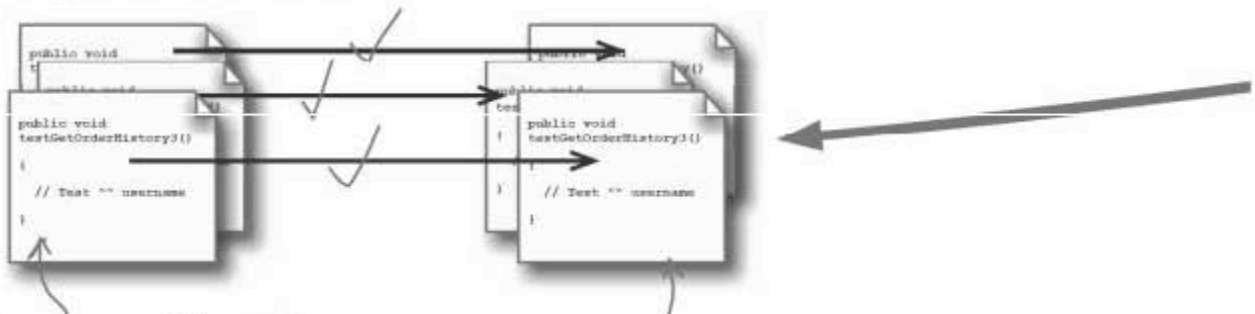
Then the automated mail starts....

1 Start with the task you're going to work on.



5 Write code to get your test to pass, refactor, add another test, and get it to pass. Repeat until you run out of tests to add.

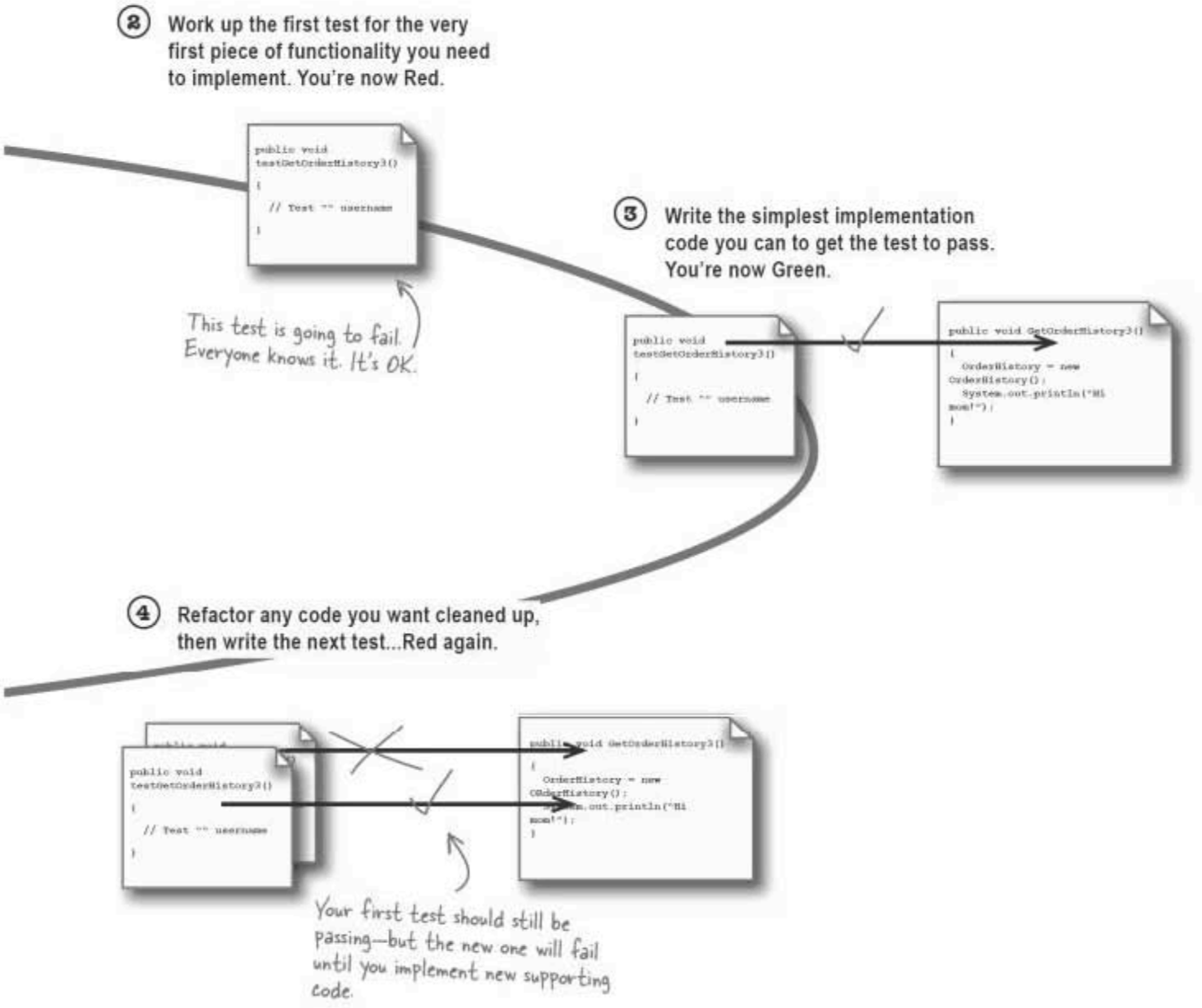
Once your code is passing, it's time to check it into your repository and move onto the next task.



All your tests should be passing (and your CI tool should make sure they stay passing).

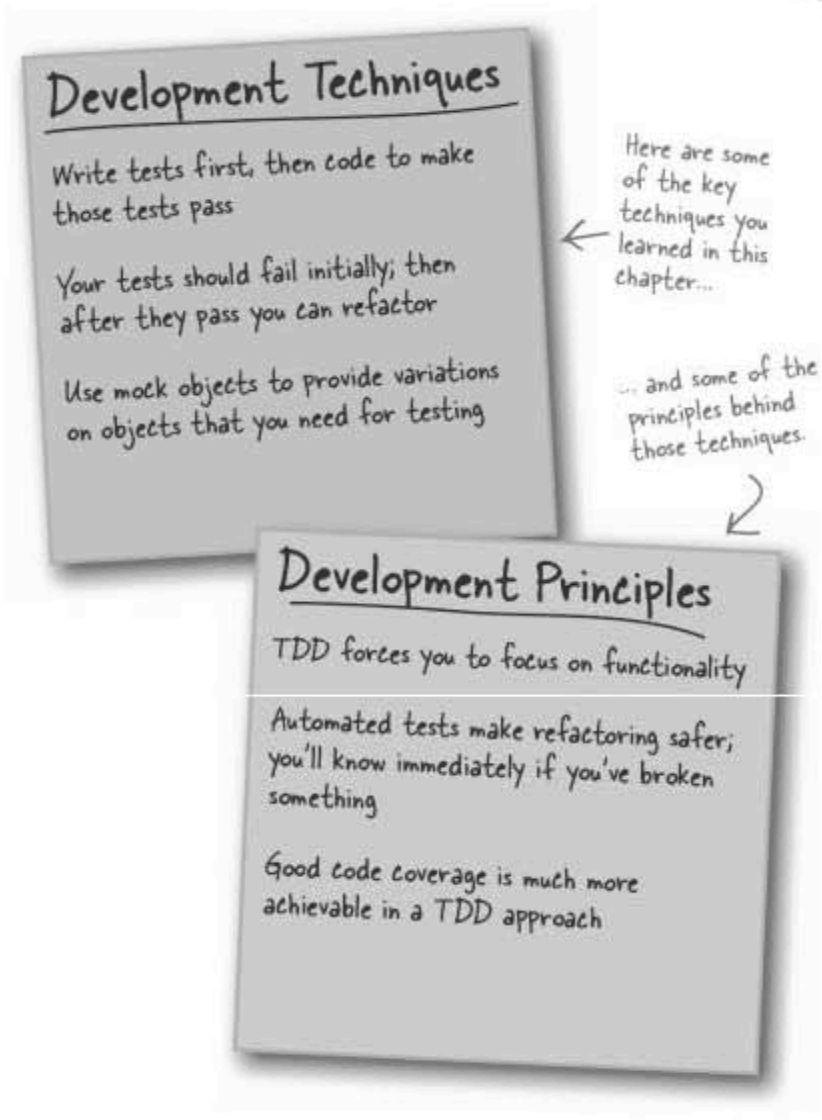
Real implementation code

* but in fact, with TDD, Bob will know instantly because the tests will fail and he will know exactly what code he broke.


2 Work up the first test for the very first piece of functionality you need to implement. You're now Red.
public void testGetOrderHistory3()
{
// Test "" username
}
This test is going to fail. Everyone knows it. It's OK.
3 Write the simplest implementation code you can to get the test to pass. You're now Green.
public void testGetOrderHistory3()
{
// Test "" username
}
public void GetOrderHistory3()
{
OrderHistory = new OrderHistory();
System.out.println("Hi mom!");
}
4 Refactor any code you want cleaned up, then write the next test...Red again.
public void testGetOrderHistory3()
{
// Test "" username
}
public void GetOrderHistory3()
{
OrderHistory = new OrderHistory();
System.out.println("Hi mom!");
}
Your first test should still be passing—but the new one will fail until you implement new supporting code.

Tools for your Software Development Toolbox

Software Development is all about developing and delivering great software. In this chapter, you learned about several techniques to keep you on track. For a complete list of tools in the book, see Appendix ii.

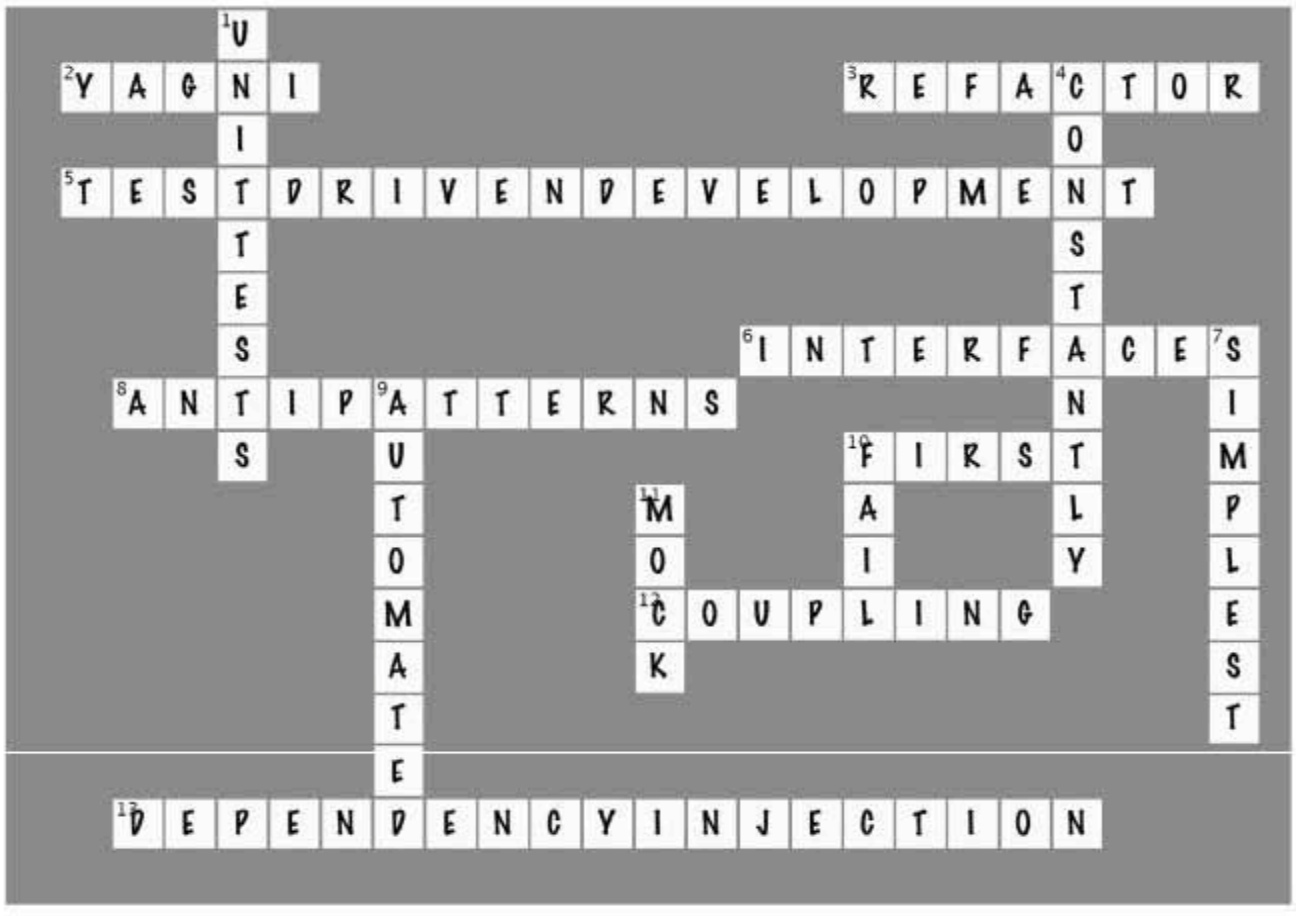


BULLET POINTS

- TDD means you'll be **refactoring code** a lot. Break something pretty bad? Just use your version control tool to roll back to where you were earlier and try again.
- Sometimes testing will **influence your design**—be aware of the trade-offs and deliberately make the choice as to whether it's worth the **increased testability**.
- Use the **strategy pattern** with **dependency injection** to help **decouple classes**.
- Keep your tests in a **parallel structure** to your source code, such as in a `tests/` directory. Most build and automated testing tools play nicely with that setup.
- **Try to keep your build and test execution time down** so running the full suite of tests doesn't hold back your development speed.

TDDcross Solution

The crossword tests are below—fill in the answers to make each one pass.

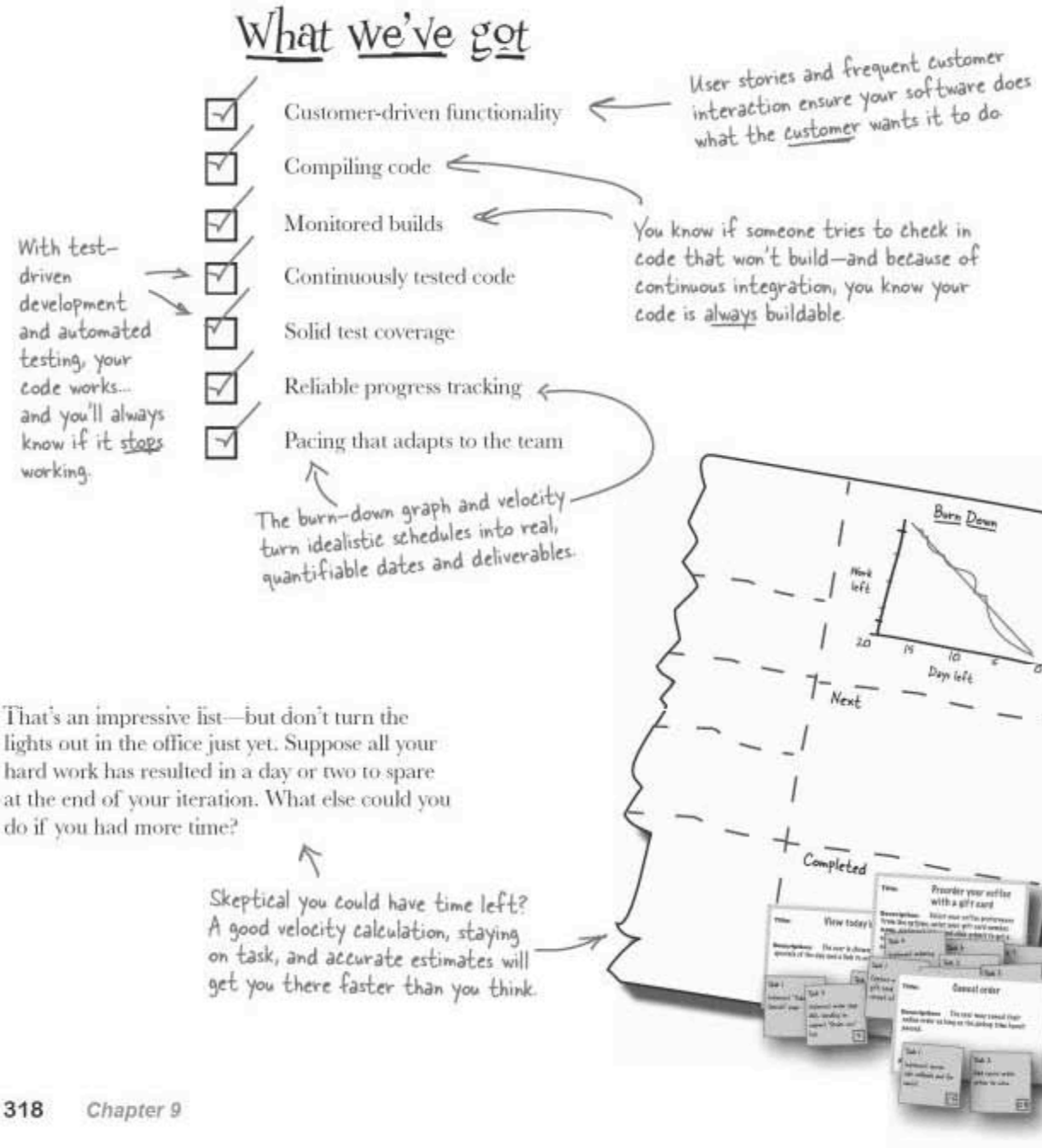


It's all coming together...

You're almost finished! The team's been working hard, and things are wrapping up. Your tasks and user stories are **complete**, but what's the best way to spend that extra day you ended up with? Where does **user testing** fit in? Can you squeeze in one more round of **refactoring** and **redesign**? And there sure are a lot of lingering **bugs**...when do those get fixed? It's all part of **the end of an iteration**... so let's get started on getting finished.

Your iteration is just about complete...

You've made it! You've successfully put your process in place: the stories have piled up in the Completed section of your board, and everyone's ready for a little breather. Before people head out for the weekend, though, let's do a quick status check:



That's an impressive list—but don't turn the lights out in the office just yet. Suppose all your hard work has resulted in a day or two to spare at the end of your iteration. What else could you do if you had more time?

Skeptical you could have time left? A good velocity calculation, staying on task, and accurate estimates will get you there faster than you think.

...but there's lots left you could do

There are always more things you can do on a project. One benefit of iterative development is that you get a chance to step back and think about what you've just built every month or so. But lots of the time, you'll end up wishing you'd done a few things differently. Or, maybe you'll think of a few things you wish you could ***still*** do...

What we don't have

- ☐ Process improvements
- ☐ System testing
- ☐ Refactoring of code using lessons you've learned
- ☐ Code cleanup and documentation updates
- ☐ More design patterns?
- ☐ Development environment updates
- ☐ R&D on a new technology you're considering
- ☐ Personal development time to let people explore new tools or read

You've worked hard putting this process together, but the whole point of iterative development is to learn from each iteration... how can you improve your processes on the next iteration?

You've got unit tests, but users haven't tried the system out yet. And users always find things the best testers miss...

Everyone documented their code, right? No typos, misspelliins, or incomplete ?

No matter how slick your design seemed early on, you'll always come up with just a little something that will make it so much sweeter. Do you do that now?

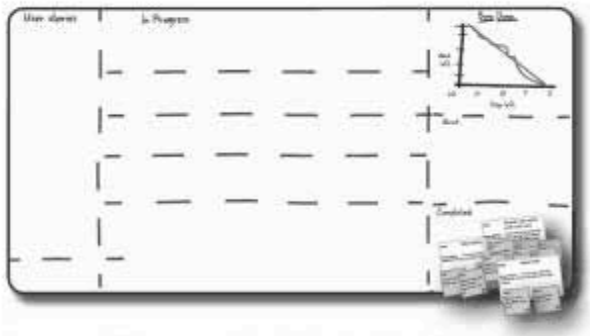
Sometimes a design pattern doesn't really show itself until you've implemented something more than once. Maybe you didn't need a factory in the first iteration... or the second. But by the time you add more code in the third iteration, things are screaming for a helpful pattern.

There's always some new tool out there that will "revolutionize" your build environment—or maybe you just need to reorganize dependencies. Either way, when do you update your environment?

You may be cutting-edge now, but when do you have time to learn about even newer technologies and work them into your projects?

Which of these things would you feel like you **have** to do? Which ones do you think you **should** do? Are there things that can be put off indefinitely? Are there other things you'd like to do that are not on this list?

Standup Meeting



Laura: OK, my code's all checked in. But I need another couple days to refactor it. A way better design came to me last night at the gym!

Mark: No way. Have you seen some of the documentation Bob put in there? I mean, it's English, I guess, but it needs some work. So no time for more code changes; we've got to work on the documentation.

Bob: Hey—back off. It says what the code does, right? Besides, we really need to test more. Everybody's tests pass, but I'm just not convinced the user isn't going to get confused navigating through some of the site's pages. And I'd like to run the app for at least a day straight, make sure we're not chewing up resources somewhere.

Laura: But we're going to have to add more complex ordering in the next iteration; the current framework just isn't going to hold up. I need to get in there and sort this out before we build more on top of it.

Mark: Are you listening? The documentation's ***awful***; that's got to be the priority with the time we've got left.

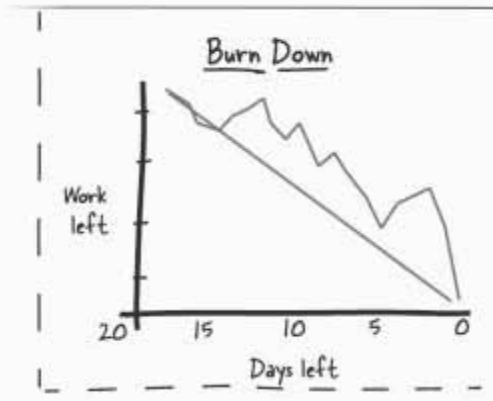
Bob: We need to focus on the project—how did our burn-down rate look this iteration? Where did we spend our time?

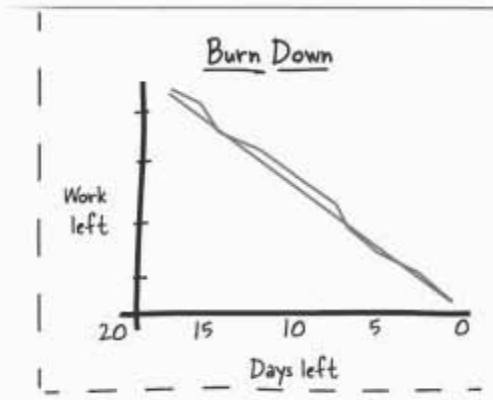
Standup Meeting Tips for Pros

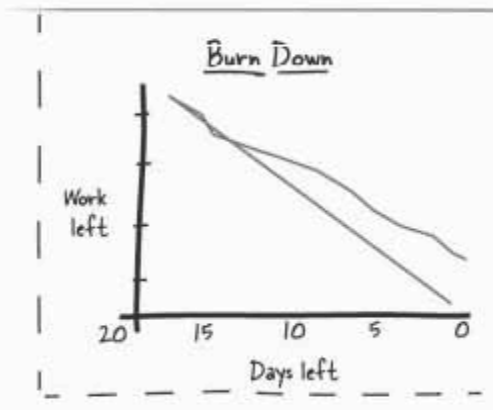
- Keep them to 10 people or less.
- Literally stand up to help keep them short—ideally 15 minutes or less, 30 if you absolutely have to, but then kick everyone out.
- Meet at the same time, same place, every day, ideally in the morning, and make them mandatory.
- Only people with direct, immediate impact on the progress of the iteration should participate; this is typically the development team and possibly a tester, marketing, etc.
- Everyone must feel comfortable talking honestly: standups are about communication and bringing the whole team to bear on immediate problems.
- Always report on *what you did yesterday*, *what you're going to do today*, and *what is holding you up*. Focus on the outstanding tasks!
- Take things offline to solve bigger issues—remember, 15 minutes.
- Standups should build the sense of team: be supportive, solve hard issues offline, and communicate!

Exercise

Do you think the tasks you'd do at the end of an iteration should be changed based on how the iteration progressed? Below are three different burn-down graphs. Can you figure out how the iteration went in each case? Describe what you think happened in the provided blanks.

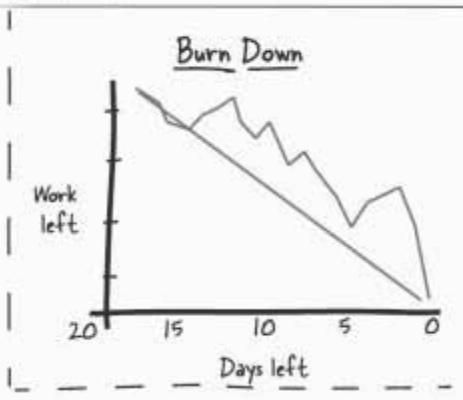




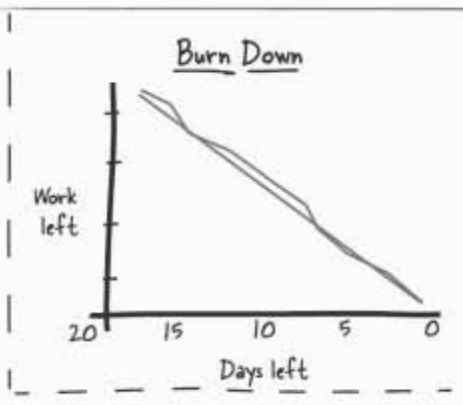


Exercise Solution

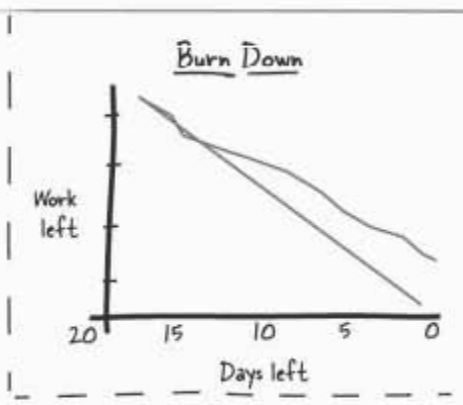
Before we go on let's take a look at some burn rates. Your job was to take a look at each graph and figure out what probably went on during that iteration.



In this graph, the work remaining kept increasing as the iteration progressed. The team probably missed some things in their user stories: maybe lots of unplanned tasks–remember, red stickies are great for those–or bad estimates that got uncovered when user stories were broken down into tasks. Note the steep drop at the end—odds are that the team had to cut out things, or drop stories altogether, as deadlines started creeping up.



This is a perfect graph—what every team wants. The team probably had a good idea what they were getting into, their estimates were pretty close at both the user story and task level, and they moved through tasks and stories at a nice predictable pace. Remember, a good iteration doesn't have lots of time at the end—it ends right when it's scheduled to.



In this graph, the work left just keeps drifting to the right of the ideal burn-down rate. Chances are this is an estimate problem. There aren't any real spikes in the work left, so it's not likely that there were too many things the team didn't account for, but they just severely underestimated how long things would take. Notice they didn't make it to zero here...The team probably should have dropped a few stories to end the iteration on time.

there are no Dumb Questions

Q: How do you know the first graph is things the team missed? Couldn't it be things they didn't expect, like extra demos or presentations?

A: Absolutely. The burn-down graph isn't enough to go on to determine where all those extra work items came from. You need to look at the completed tasks and figure out whether the extra work came from outside forces that you couldn't control or if they were a result of not really understanding what the team was getting into. Either way, it's important to make progress in addressing the extra work before the next iteration. If the work came from outside sources, can you do something to limit that from happening again, or at least incorporate it into your work for the estimate? For example, if the marketing team keeps asking you for demos, can you pick one day a week where they could get a demo if needed? You can block that time off and count it toward the total work left. You can use the same approach if things like recruiting or interviewing candidate team members is taking time away from development. Remember—your job is to do ***what the customer wants***. However, it's also your responsibility to know where your time is going and prioritize appropriately.

If the extra work came from not understanding what you were getting into, do you have a better sense now, after working on the project for another iteration? Would spending more time during task breakdowns help the team get a better sense of what has to be done? Maybe some more up-front design, or possibly quick-and-dirty code (called spike implementations) to help shake out the details?

Q: So spending more time doing up-front design usually helps create better burn-down rates, right?

A: Maybe...but not necessarily. First, remember that by the time you start doing design, you're *already* into your iteration. Ideally you'd find those issues earlier.

It's also important to think about ***when*** is the right time to do the design for an iteration. Some teams do most of the detailed design work at the beginning of the iteration to get a good grasp of everything that needs to be done. That's not necessarily a bad approach, but keep an eye on how efficient you are with your designs. If you had driven a couple stories to completion before you worked up designs for some of the remaining ones, would you have known more about the rest of the iteration? Would the design work have gone faster, or would you realize things you'd need to go back and fix in the first few stories? It's a trade off between how much up-front design you do before you start coding.

Having said all of that, sometimes doing some rough whiteboard design sketches and spending a little extra time estimating poorly understood stories can help a lot with identifying any problem issues.

Q: For that third graph, couldn't the velocity be a big part of the problem?

A: That's a possibility, for sure. It could either be that the team's estimates were wrong and things just took a lot longer than they thought the would, or their estimates were reasonable but they just couldn't implement as fast as they thought. At the end of the day it doesn't make too much difference. As long as a team is consistent with their estimates, then velocity can be tweaked to compensate for over- or underestimating. What you ***don't*** want to do is keep shifting your estimates around. Keep trying to estimate for that ***ideal workday for your average developer***—if that person was locked in a room with a computer and a case of Jolt, how long would it take? Then, use velocity to adjust for the reality of your work environment and mixed skill level on your team.

Q: So should the team with the third graph just add time to the end of their iteration to get the extra work done?

A: In general that's not a great idea. Typically, when the burn-down graph looks like that, people are already working hard and feeling stressed. Remember one of the benefits of that graph on the board is communication—everyone sees it at each standup, and they know things are running behind. Adding a day or two is usually OK in a crisis, but not something you want to do on a regular basis. Adding a week or two... well, unless it's your last iteration, that's probably not a good idea. It's generally better to punt on a user story or two and move them to the next iteration. Clean up the stories you finished, get the tests passing, and let everyone take a breather. You can adjust your velocity and get a handle on what went wrong before you start the next iteration, and go into it with a refreshed team and a more realistic pace.

Q: We have one guy who just constantly underestimates how long something is going to take and wrecks our burn-down. How do we handle that?

A: First, try to handle the bad estimates during estimation, and remember, you should be estimating as a team. Try reminding the person that they aren't estimating for themselves, but for the average person on your team. If that still doesn't work, try keeping track of the date a task gets moved to In Progress, and then the date it gets moved to Done. At the end of your iteration, use that information to calibrate your estimations. Remember, this isn't about making anyone feel bad because they took longer than originally expected; it's to calibrate your estimates from the beginning.

System testing MUST be done...

Your system has to work, and that means **using the system**. So you've got to either have a dedicated end-to-end system testing period, or you actually let the real users work on the system (even if it's on a beta release). No matter which route you go, you've got to test the system in a situation that's as close to real-world as you can manage. That's called **system testing**, and it's all about reality, and the system as a whole, rather than all its individual parts.

So far, we've been **unit testing**. Our tests focus on small pieces of code, one at a time, and deliberately try to isolate components from each other to minimize dependencies. This works great for automated test suites, but can potentially miss bugs that only show up when components interact, or when real, live users start banging on your system.

And that's where **system testing** comes in: hooking everything together and treating the system like a black box. You're not thinking about how to avoid garbage collection, or creating a new instance of your `RouteFinder` object. Instead, you're focusing on the functionality the customer asked for... and making sure your system handles that functionality.

System testing exercises the FUNCTIONALITY of the system from front to back in real-world, black-box scenarios.

...but WHO does system testing?

You should try your best to have a **different set of people** doing your system testing. It's not that developers aren't really bright people; it's just that dedicated testers bring a testing mentality to your project.

Developer testing

Developers come preloaded with lots of knowledge about the system and how things work underneath. No matter how hard they try, it's really tough for developers to ***put themselves in the shoes of end users*** when they use the system. Once you've seen the guts, you just can't go back.

Tester testing

Testers can often bring a fresh perspective to the project. They approach the system with a fundamentally different view. They're *trying* to find bugs. They don't care how slick your multithreaded, templated, massively parallel configuration file parser is. ***They just want the system to work.***

there are no Dumb Questions

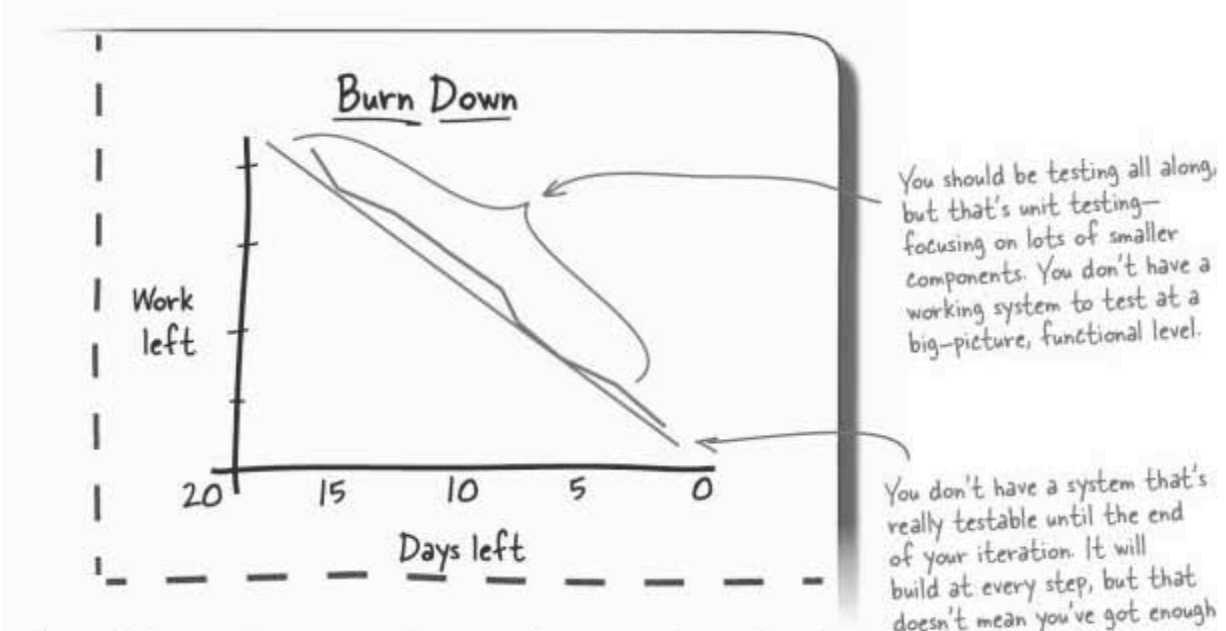
Q: So developers can't be testers? We can't afford a separate test team!

A: Ideally, you'd have developers doing your unit testing with an automated approach, and a different group of people doing the full, black-box system testing. But, if that's not doable, then ***at a minimum***, don't let a developer black-box-test their own code. They just know too much about the code, and it's way too easy to steer clear of that sketchy part of the code that just might fail.

Never system-test your own code! You know it too well to be unbiased.

System testing depends on a complete system to test

If you're velocity is pretty accurate and your estimates are on, you should have a reasonably full iteration. It also means you don't have a stack of empty days for system testing...and on top of that, you won't have a system to test until the end of your iteration.



At a minimum, the system needs to get out for system testing at the end of each iteration. The system won't have all of its functionality in the early iterations, but there should always be some completed stories that can be tested for functionality.

there are no Dumb Questions

Q: Can't we start system testing earlier?

A: Technically, you can start system testing earlier in an iteration, but you really have to think about whether that makes much sense. Within an iteration, developers often need to refactor, break, fix, clean up, and implement code. Having to deliver a build to another group in the middle of an iteration is extremely distracting and likely to including half-baked features. You also want to try to avoid doing bug fix builds in the middle of an iteration—an iteration is a fixed amount of time the team has to make changes to the system. They need to have the freedom to get work done without worrying about what code goes in which build during the iteration. Builds get distributed at the end of an iteration—protect your team in between!

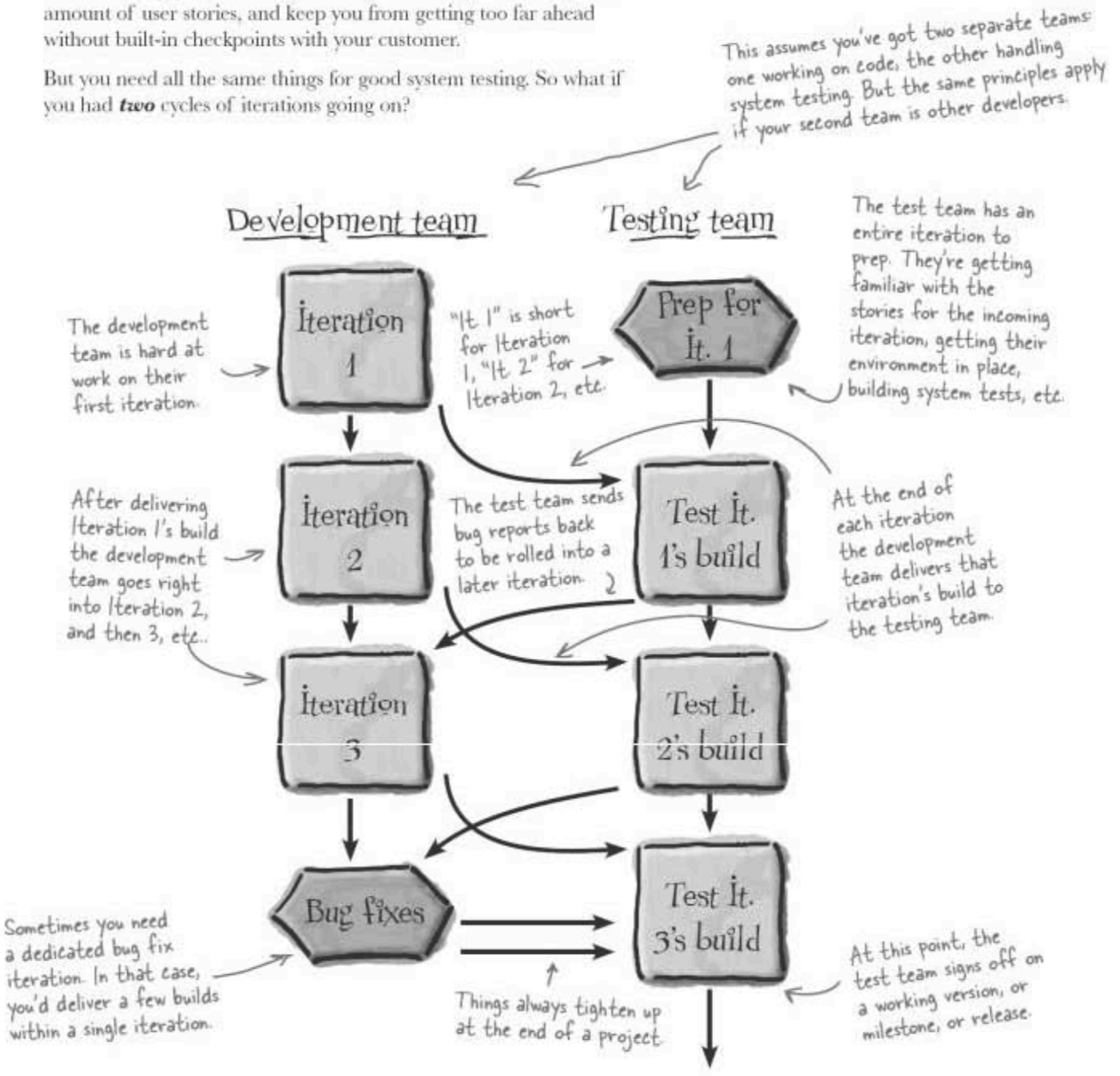
Q: So what about the people doing testing? Where do they fit in?

A: It's definitely best to have a separate group doing system testing, but as for what they should do while your main team is writing code, that's a good question. And even if you have other developers do system testing, the question still applies...

Good system testing requires TWO iteration cycles

Iterations help your team stay focused, deal with just a manageable amount of user stories, and keep you from getting too far ahead without built-in checkpoints with your customer.

But you need all the same things for good system testing. So what if you had ***two*** cycles of iterations going on?



More iterations means more problems

System testing works best with two separate teams, working two separate iteration cycles. But with more iterations comes more trouble—problems that aren't easy to solve.

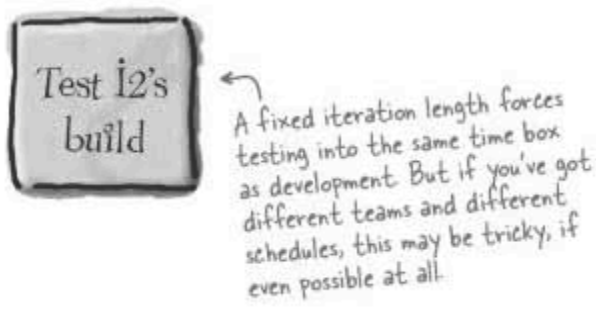
Running two cycles of iterations means you've got to deal with:

LOTS more communication

Now, not only do you have inter-team communication issues, but you've got two teams trying to work together. The testing team will have questions on an iteration, especially about error conditions, and the development team wants to get on to the next story, not field queries. One way to help this is to bring a representative from the test team into your standup meetings as an observer. He'll get a chance to hear what's going on each day and get to see the any notes or red stickies on the board as the iteration progresses. Remember that your standup meeting is ***your*** meeting, though—it's not a time to prioritize bugs or ask questions about how to run things.

Testing in a FIXED iteration length

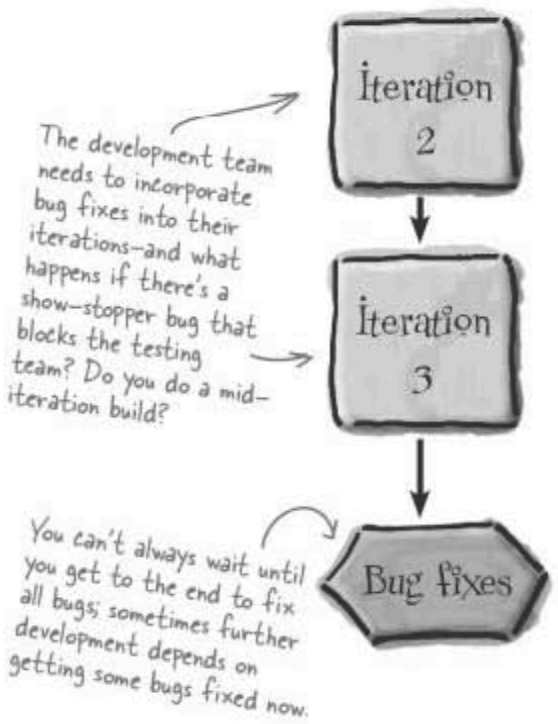
If you're keeping your two iteration cycles in sync—and that's the best way to keep the testing team caught up—you're forcing testing to fit into a length that might not be ideal. To help give the test team a voice in iterations, you can have them provide you a testing estimate for stories you're planning on including in your iteration. Even if you don't use that to adjust what's in your iteration (remember, you're priority-driven) it might give you some insight into where the testing team might get hung up or need some help to get through a tough iteration.



Fixing bugs while you keep working

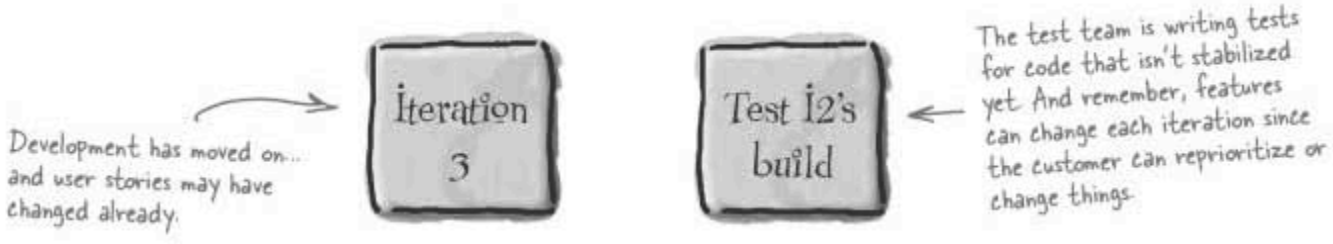
The development team will start getting bug reports on their first iteration about the time they're getting into the ***third*** iteration! And then you have to figure out if the bug's important enough to fix right away, roll into the current iteration, or put off for later. We'll talk more about this in a minute, but the straightforward approach is to treat a bug like any other story. Prioritize it against everything else and bump lower-priority stories if you need to in order to get it done sooner.

Another approach is to carve off a portion of time every week that you'll dedicate to bug fixes. Take this off of the available hours when you do your iteration planning, so you don't need to worry about it affecting your velocity. For example, you could have everyone spend one day a week on bug fixes—about 20% of their time.



Writing tests for a moving target

Functionality in user stories—even if it's agreed upon by the customer—can change. So lots of times, tests and effort are being put into something that changes 30 days later. That's a source of a lot of irritation and frustration for people working on tests. There's not much you can do here except **communicate**. Communicate as soon as you think something might change. Make sure testing is aware of ongoing discussions or areas most likely to be revisited. Have formal turnover meetings that describe new features and bug fixes as well as known issues. One subtle trick that people often miss is to *communicate how the process works*. Make sure the testing team understands to expect change. It's a lot easier to deal with change if it's just part of your job rather than something that's keeping you from completing your job.



More iterations really just means more communication.

During an iteration there are some messy things to deal with: multiple team members, your customer changing requirements and user stories, priorities of different pieces of functionality, and sometimes having to guess at what you're going to build before your requirements are complete.

Adding another cycle of iterations might mean more of the same issues, but you won't have any new ones. That means you can rely on the same things you've already been doing: standup meetings, tracking everything you do on your big board, using velocity to account for real life, and lots and lots of communication—with your team, with the testing team, and, of course, with your customer.

The key to most problems you'll run into in software development is COMMUNICATION.

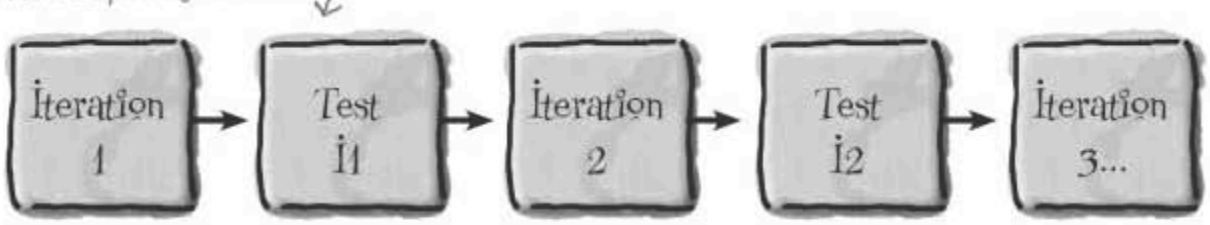
When in doubt, TALK to your team, other teams, and your customer.

Sharpen your pencil

Below are some different approaches to testing, all of which involve just one cycle of iterations. What are some **good things** about these approaches? What are some **bad things**?

This approach has one big testing iteration at the end.

This approach adds a testing iteration after every coding iteration.



Sharpen your pencil Solution

Below are some different approaches to testing, all of which involve just one cycle of iterations. What are some **good things** about these approaches? What are some **bad things**?

If you only have one team to work with, this approach isn't too bad. One big drawback is that serious system testing starts very late in the process. If you take this approach, it's critical that the results of each iteration get out to at least a set of beta users and the customer. You can't wait until the end of the third iteration to start any testing and collecting feedback.

This approach also works pretty well if you need to do formal testing with the customer before they sign off on your work. Since you've been doing automated testing during each iteration and releasing your software to users at the end of each iteration you have a pretty good sense that you're building the right software and it's more or less working as expected. The test iteration at the end is where the formal "check-off" happens before you start looking at Version 2.0.

This is usually called acceptance testing.

This approach requires a lot of iterations, and 50% of your time is spent in testing. It really would only work in situations where your customer is willing to expend a lot of time on testing and debugging. Let's say that your customer is thrilled with the idea of monthly releases to the public; it keeps the site fresh and dynamic in their users' eyes. However, the customer insists on a formal validation process before the code goes anywhere. If you don't have a separate acceptance- and system-testing team, you're going to be looking at a situation a lot like this.

Top 10 Traits of Effective System Testing

10 **Good, frequent communication** between the customer, development team, and testing team.

9 **Know the starting and ending state of the system.** Make sure you start with a known set of test data, and that the data ends up exactly like you'd expect it at the end of your tests.

8 **Document your tests.** Don't rely on that one awesome tester who knows the system inside and out to always be around to answer questions. Capture what each tester is doing, and do those same things at each round of system testing (along with adding new tests).

7 **Establish clear success criteria.** When is the system good enough to go live? Testers can test forever—know before you start what it means to be finished. A **zero-bug-bounce** (when you get to zero outstanding bugs, even if you bounce back up after that) is a good sign you're getting close.

6 **Good, frequent communication** between the customer, development team, and testing team.

5 **Automate your testing wherever possible.** People just aren't great at performing repetitive tasks carefully, but computers are. Let the testers exercise their brains on new tests, not on repeating the same five over and over and over again.

4 **A cooperative dynamic between the development team and testing team.** Everyone should want solid, working software that they can be proud of. Remember, testers help developers look good.

3 **A good view of the big picture by the testing team.** Make sure that all your testers understand the overall system and how the pieces fit together.

2 **Accurate system documentation** (stories, use cases, requirements documents, manuals, whatever). In addition to testing docs, you should capture all of the subtle changes that happen during an iteration, and especially *between* iterations.

1 **Good, frequent communication** between the customer, development team, and testing team.

The life (and death) of a bug

Eventually, your testers are going to find a bug. In fact, they'll probably find a lot of them. So what happens then? Do you just fix the bug, and not worry about it? Do you write it down? What really happens to bugs?

1 **A tester FINDS A BUG**
A bug doesn't have to be something that's clearly failing. It could be ambiguity in the documentation, a missing feature, or a break from the style guide for a web site.

Just like with version control and building, there are great tools for tracking and storing bugs.

2 **The tester FILES A BUG REPORT**
This is one of the most critical steps: ***you have to track bugs!*** It doesn't matter who reports a bug, but level of detail is crucial. Always record what you were trying to do, and if possible, the steps to re-create the bug, any error messages, what you did immediately before the bug occurred, and what you would have expected to happen.

6 **UPDATE the bug report**
Once the tester (and original reporter) are happy with the fix, close the bug report. The updated report can be used as a script to retest. Don't delete it...you never know when you might want to refer back to it.

Some teams work with their customer to prioritize which bugs get fixed using the bug tracker, and don't create stories or tasks for ones that aren't going to get fixed in the current iteration. Other teams create the stories and tasks right away and let the customer prioritize the bugs like any other story. Either way works great as long as things are being prioritized by the customer.

3 **CREATE A STORY (or task) to fix the bug**

Bugs are just work that has to be done in your system—sometimes in the current iteration, sometimes in a later one. You'll need to capture them and prioritize each bug with the customer. These are tricky to estimate, though, because it's not always clear what's wrong. Some teams have a "Bug Fix" story that they just keep around, and they add tasks to it as needed.

A bug is just like an unplanned task. Once it's on the board, it's handled like any other story and task.

4 **FIX THE BUG**

The development team works on the bug as part of an iteration. Start by writing a test that exposes the bug (the test should fail before you change any code). Once the team's fixed the bug (and the test lets you know when that is), they should mark it as "Fixed" in the bug tracker. But don't mark it as tested, closed, or verified—that's for the original reporter to take care of. This also helps you get a list of what's ready for turnover to the test team.

A build usually has more than just one bug fix in it, but it can depend on how critical the bug was.

5 **CHECK THE FIX and verify it works**

The tester (or original reporter) verifies the new build and makes sure they're happy with the resolution. *Now* the bug can be marked as closed (or verified).

So you found a bug....

No matter how hard you work at coding carefully, some bugs are going to slip through. Sometimes they're programming errors; sometimes they're just functional issues that no one picked up on when writing the user stories. Either way, a bug is an issue that you ***have*** to address.

Bugs belong in a bug tracker

The most important thing about dealing with bugs on a software project is making sure they get recorded and tracked. For the most part it doesn't matter which bug tracking software you use; there are free ones like Bugzilla and Mantis or commercial ones like TestTrackPro and ClearQuest. The main thing is to make sure the whole team knows how to use whatever piece of software you choose.

You should also use your tracker for more than just writing down the bug, too. Make sure you:

1. **Record and communicate priorities**
   Bug trackers can record priority and severity information for bugs. One way to work this in with your board is to pick a priority level—say priority 1, for example—and all bugs of that priority level get turned into stories and prioritized with everything else for the next iteration. Any bugs below priority 1 stay where they are until you're out of priority 1 bugs.

   Bug trackers usually work off priorities like 1, 2, and 3, even though your user stories have priorities more like 10, 20, and 30.

2. **Keep track of everything**
   Bug trackers can record a history of discussion, tests, code changes, verification, and decisions about a bug. By tracking everything, your entire team knows what's going on with a bug, how to test it, or what the original developer thought they did to fix it.

3. **Generate metrics**
   Bug trackers can give you a great insight into what's really going on with your project. What's your new-bug submission rate? And is it going up or down? Do a significant number of bugs seem to come from the same area in the code? How many bugs are left to be fixed? What's their priority? Some teams look for a **zero-bug-bounce** before even discussing a production release; that means all of the outstanding bugs are fixed (bug count at zero) before a release.

   We'll talk more about delivering software in Chapter 12.

Anatomy of a bug report

Different bug tracking systems give you different templates for submitting a bug, but the basic elements are the same. As a general rule of thumb, the more information you can provide in a bug report, the better. Even if you work on a bug and don't fix it, you should record what you've done, and any ideas about what else might need to be done. You—or another developer—might save hours by referring to that information when you come back to that bug later.

A good bug report should have:

- ☐ **Summary:** Describe your bug in a sentence or so. This should be a detailed action phrase like "Clicking on received message throws ArrayOutOfBoundsException," not something like "Exception thrown." You should be able to read the summary and have a clear understanding of what the problem is.

- ☐ **Steps to reproduce:** Describe how you got this bug to happen. You might not always know the exact steps to reproduce it, but list everything you think might have contributed. If you can reproduce the bug, then explain the steps in detail:
  1. Type "test message" into message box.
  2. Click "sendIt."
  3. Click on the received message in the second application.

- ☐ **What you expected to happen and what really did happen:** Explain what you thought was going to happen, and then what actually *did* happen. This is particularly helpful in finding story or requirement problems where a user expected something that the developers didn't know about.

- ☐ **Version, platform, and location information:** What version of the software were you using? If your application is web-based, what URL were you hitting? If the app's installed on your machine, what kind of installation was it? A test build? A build you compiled yourself from the source code?

- ☐ **Severity and priority:** How bad is the impact of this bug? Does it crash the system? Is there data corruption? Or is it just annoying? How important is it that the bug gets fixed? Severity and priority are often two different things. It's possible that something is severe (kills a user's session or crashes the application) but happens in such a contrived situation (like the user has to have a particular antivirus program installed, be running as a non-Administrator user, and have their network die while downloading a file) that it's a low-priority fix.

What else would you want to see in a bug report? What kind of information would you want to see from the user? How about any kind of output from the system?

But there's still plenty left you COULD do...

So you've handled system testing and dealt with the major bugs you wanted to tackle this iteration. Now what?

What we don't have

- [ ] Process improvements
- [x] System testing
- [ ] Review the iteration for what worked and what didn't
- [ ] Refactor code using lessons you've learned
- [ ] Code cleanup and documentation updates
- [ ] More design patterns?
- [ ] Development environment updates
- [ ] R&D on a new technology you're considering
- [ ] Personal development time to let people explore new tools or read

System testing is taken care of, and you've knocked out your bug reports (or you're waiting on some to get filed).

The right thing to do at any time on your project is the right thing to do AT THAT TIME on YOUR project.

There are no hard-and-fast rules—you've got to make this decision yourself.

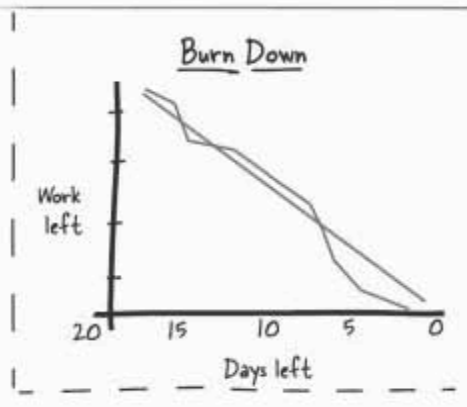
A good software process is all about **prioritization**. You want to make sure you're doing the right thing on the project at all times.

Producing working software is critical, but what about quality code? Could you be writing even better code if your process was improved? Or if you dropped a couple thousand lines by incorporating that new persistence framework?

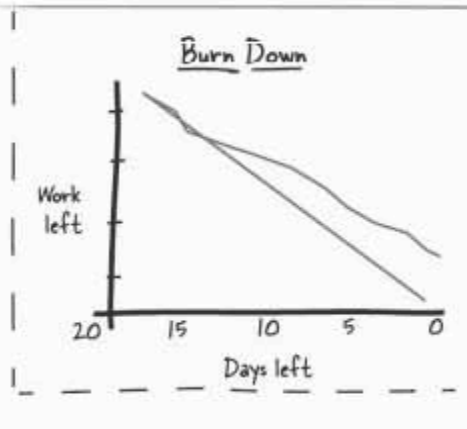
Exercise

Below are three burn-down graphs. It's up to you to decide what to do next.

What would you do next?



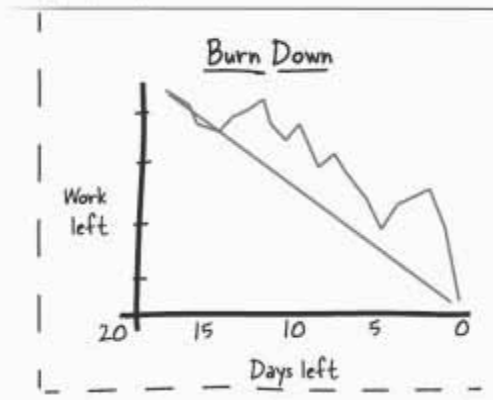
What would you do next?



What would you do next?

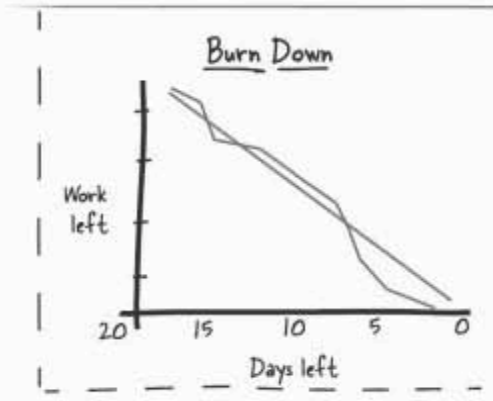
Exercise Solution

Below are three burn-down graphs. It's up to you to decide what to do next.



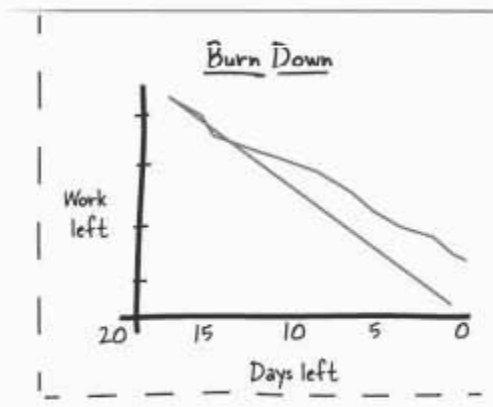
What would you do next?

Here, the team just got finished at the end of the iteration, so there's likely nothing you can squeeze in. However, that steep drop at the end probably means something was skipped. Testing is going to be vital after this iteration, and you should probably expect to schedule some time next iteration for refactoring and cleanup, to recover from the rush. You could probably revisit your task breakdown approach, too, as well as take a look at adjusting velocity.



What would you do next?

In this iteration, things wrapped up early, the team may have a couple of days at the end of the iteration. If the project has been ticking along for a little bit, you may have a backlog of bugs you can start to tick off. Or, depending on how big your stories are, you might be able to grab the highest priority story waiting for the next iteration and get started on that.



What would you do next?

Nothing! This team is late already. Before the next iteration, you should look at what caused the slowdown and whether it's a velocity problem, an estimation problem, or something else. Chances are there's unfinished code, too, which means there are going to be bugs coming your way. Make sure you leave room in the next iteration to cover any problems that came up.

How can you decide what to do next when you haven't even seen the code the team is working on?

Each project *is* different...sort of.

Remember, software development isn't about just code. It's about good habits and approaches to deliver working software, on time and on budget. Besides, we already have:

- ☑ Compiling code
- ☑ Automated unit testing
- ☑ User stories and tasks that capture what needs to happen
- ☑ A process for prioritizing what gets done next
- ☑ A working build of the software we can deliver to the customer

So this is about how to prioritize additional, nice-to-have tasks, if you've got extra time. And that's all about ***where you are in your project.*** Early on you'll likely need more refactoring to refine your design. Later, when the project is a little more stable, you'll probably spend more time on documentation or looking at alternatives to that aging technology the team started with six months ago.

Time for the iteration review

It's here: the end of your iteration. You're remaining work is at zero, you've hit the last day of the iteration, and it's time to start getting ready for the next iteration.

But, before you prioritize your next stories, remember: it's not just software we're developing iteratively. You should develop and define your process iteratively, too. At the end of each iteration, take some time to do an iteration review with your team. Everyone has to live with this process so make sure you incorporate their opinions.

Elements of an iteration review

1. **Prepare ahead of time**
An iteration review is a chance for the team to give you their input on how the iteration went, not a time for you to lecture. However, it's important to keep the review focused, too. Bring a list of things you want to make sure get discussed and introduce them when things start wandering off.

2. **Be forward-looking**
It's OK if the last iteration was tragic or if one if the developers consistently introduced bugs, as long as the team has a way to address it in the next iteration. People need to vent sometimes, but don't let iteration reviews turn into whining sessions; it demoralizes everyone in the end.

3. **Calculate your metrics**
Know what your velocity and coverage were for the iteration that just completed. In general, it's best to add up all of the task estimates and divide by the theoretical person-days in your iteration to get your velocity. Whether or not you reveal the actual number during the review is up to you (sometimes it helps to not give the actual number just yet so as not to bias any upcoming estimates), but you should convey whether the team's velocity went up or down.

The actual time spent on a task versus the estimated time isn't too important since your velocity calculation should account for any mismatch. We'll talk about velocity again in the next chapter.

4. **Have a standard set of questions to review**
Have a set of questions you go through at the end of each iteration. The set of questions can change if someone would like to add something or a question really doesn't make sense for your team. Having recurring discussion topics means people will expect the questions and prepare (even unconsciously during an iteration) for the review.

Some iteration review questions

Here is a set of review questions you can use to put together your first iteration review. Add or remove questions as appropriate for your team, but try to touch on each of the general areas.

Iteration Review Questions

- ☐ Is everyone happy with the quality of work? Documentation? Testing?
- ☐ How did everyone feel about the pace of the iteration? Was it frantic? Reasonable? Boring?
- ☐ Is everyone comfortable with the area of the system they were working in?
- ☐ Are there any tools particularly helping or hurting productivity? Are there any new tools the team should consider incorporating?
- ☐ Was the process effective? Were any reviews conducted? Were they effective? Are there any process changes to consider?

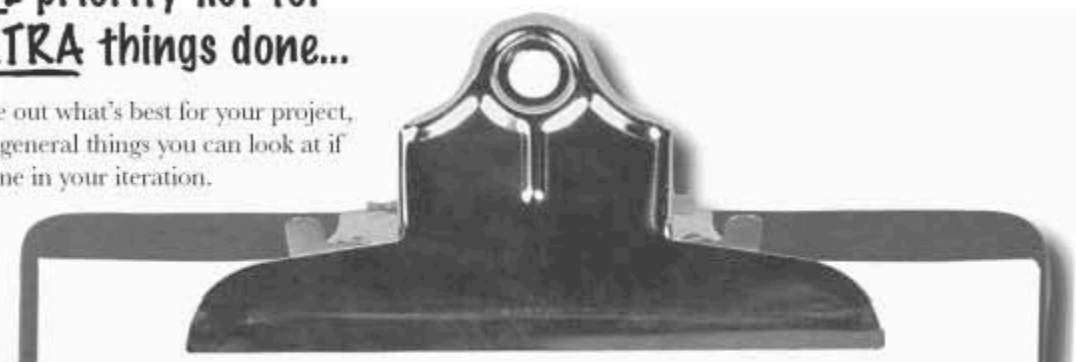
See Chapter 12 for more on this.

- ☐ Was there any code identified that should be revisited, refactored, or rewritten?
- ☐ Were any performance problems identified?
- ☐ Were any bugs identified that must be discussed before prioritization?
- ☐ Was testing effective? Is our test coverage high enough for everyone to have confidence in the system?
- ☐ Is deployment of the system under control? Is it repeatable?

Any of these questions could turn into things you'd like to get done next iteration. Remember, you should be story-driven, so make sure any changes you want to introduce support some customer need (either directly or indirectly) and get prioritized along with everything else. It might mean you need to make a case for a technology or process change to the customer, but it's important to remember why you're writing the software in the first place.

A GENERAL priority list for getting EXTRA things done...

You've got to figure out what's best for your project, but here are some general things you can look at if you've got extra time in your iteration.



Fix bugs

Obviously this depends on what your bug backlog looks like. Remember to prioritize bugs with the customer, too. There might be some bugs that are vital to the customer, and others they just don't care that much about.

Pull in stories that are on deck for next iteration

Since the customer has prioritized more stories than typically fit in an iteration, you can try pulling in a story from the next iteration and get working on it now. Be careful doing this, though, as the customer's priorities or ideas for the story may have changed during your iteration. It's also good to make sure you know whether or not the test team has time to test any extra stories you pull in.

Prototype solutions for the next iteration

If you have an idea about what's likely coming in the next iteration, you might want to take advantage of an extra day or two to start looking ahead. You could try writing some prototype code or testing technologies or libraries you might want to include. You probably won't commit this code into the repository, but you can get some early experience with things you plan on rolling into the next iteration. It will almost certainly help your estimates when you get back to planning poker.

Training or learning time

This could be for your team or for your users. Maybe the team goes to a local users group's session during work hours. Get a speaker or technology demo setup. Run a team-building exercise like sailing or paintball. **Care and feeding of your team** is an important part of a successful project.

there are no Dumb Questions

Q: Seriously, do people ever really have time at the end of the iteration?

A: Yes, absolutely! It usually goes something like this: The first iteration or two are bad news. People always underestimate how long something is going to take early on. At the end of each iteration, you adjust your velocity, so you end up fitting less into subsequent iterations. As the team gets more experienced, their estimates get better, and they get more familiar with the project. That means that the velocity from previous iterations is actually *too low*. This ends up leaving room at the end of an iteration—at least until you recalculate the team's velocity. And believe it or not, sometimes, well, things just go right and you have extra time.

Q: Wait, you said the first two iterations will be bad?

A: You don't want them to be, but realize that people almost always underestimate how long things will take—or how much time they're spending on little things that no one is thinking about, like setting up a co-worker's environment or answering questions on the user's mailing list.

Those are all important things, but need to be accounted for in your work estimates. That's part of why a velocity of 0.7 for your first iteration is a good idea. It gives you some breathing room until you really know how things are going. You'll be surprised how fast you can fill an iteration with user stories, too. Strike a balance between getting a lot squeezed in and being realistic about what you can hope to get done.

Q: We seem to always have extra time at the end of our iterations—and lots of it. What's going on?

A: One idea is that your velocity might be way off. Are you updating it at the end of each iteration? (We'll talk more about that in the next chapter) Another idea is that your estimates are off—on the high side. If you've recently had an iteration where things got really tight at the end, people will naturally be more conservative in their estimates in the next iteration. That's OK, though. If you have lots of time, pull in another story or two at the end of the iteration, and when you update your velocity it will all balance back out.

Q: We tried pulling a story into our iteration, but now it's not finished and we're just about out of time.

A: Punt on the story. Remember, you're working ahead of the curve anyway. It's better to punt on the story and put it back into the next iteration than it is to commit half-written, untested code and just "wrap that up" next iteration. Remember, you're going to send your iteration's build out into the wild. You want it as stable and clean as possible.

If there is extra time at the end of an iteration some teams will tag their code before they pull anything else into the repository. That way, if things go south, they have no problems releasing a stable build by using the tag.

Q: You keep saying to prioritize bugs... but we're in the middle of an iteration. So how do bugs fit in?

A: Some projects have regularly scheduled bug reviews with the customer once a week or so to prioritize outstanding bugs. In those cases, there's always a pool of work to pull from if there's time available. If you don't meet with your customer to talk about bugs regularly, you might want to think about that...although be sure to factor in the time you'll spend on your burn rate and big board. Remember, if the bug is sufficiently important to fix, it should get scheduled into an iteration like anything else.

It's important to note we're talking about bugs found outside of developing a story. If you find a bug in a story you're working on, you should almost always fix it (after adding a test!). Nothing is "done" until it works according to the story—and the tests are proving it.

You shouldn't have LOTS of time left over. So choose SMALL TASKS to take on with any extra time you have... and get ready for the NEXT iteration.

Tools for your Software Development Toolbox

Software Development is all about developing and delivering great software. In this chapter, you learned how to end an iteration effectively. For a complete list of tools in the book, see Appendix ii.

Development Techniques

Pay attention to your burn-down rate—especially after the iteration ends

Iteration pacing is important—drop stories if you need to keep it going

Don't punish people for getting done early—if their stuff works, let them use the extra time to get ahead or learn something new

Here are some of the key techniques you learned in this chapter...

... and some of the principles behind those techniques.

Development Principles

Iterations are a way to impose intermediate deadlines—stick to them

Always estimate for the ideal day for the average team member

Keep the big picture in mind when planning iterations—and that might include external testing of the system

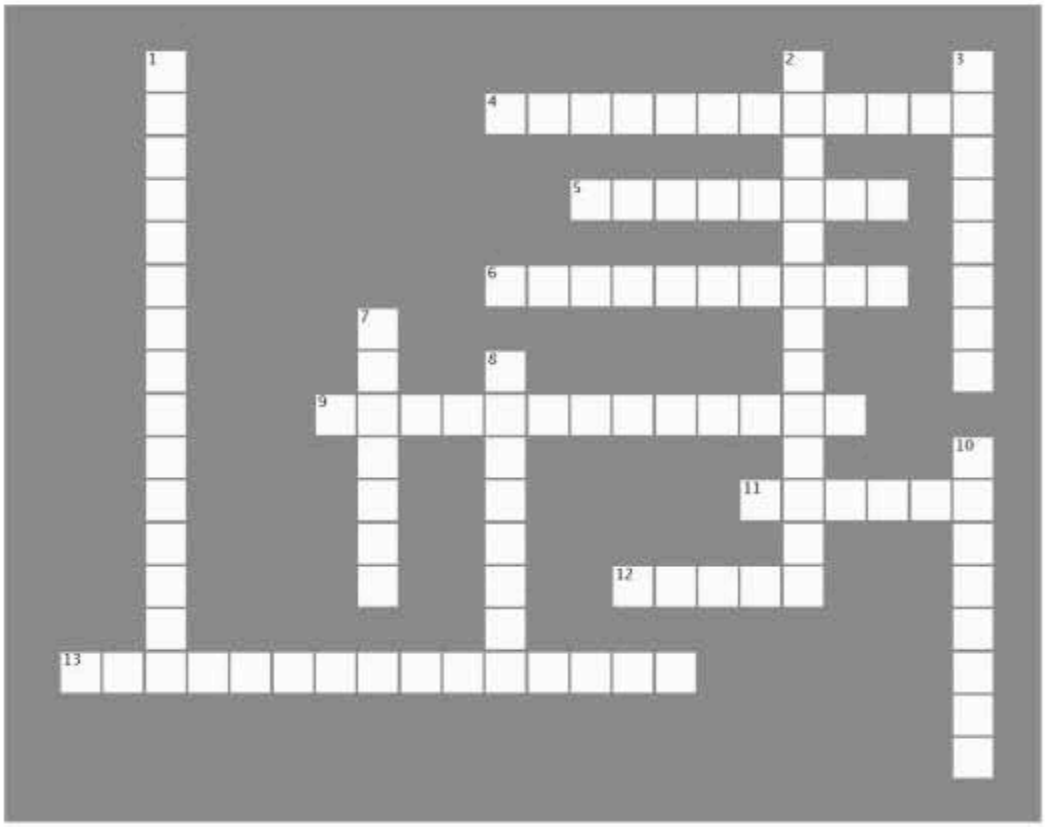
Improve your process iteratively through iteration reviews

BULLET POINTS

- If you have some room at the end of an iteration, that's a good time to **brainstorm for new stories** that might have come up. They'll need to be prioritized with everything else, but it's great to capture them.
- **Resist the temptation** to forget about all of your good habits in the last day or two of an iteration. Don't just "sneak in" that one quick feature that has a low priority because you have a day or make that little refactoring that you're "sure won't break anything." You worked really hard to get done a day or so early, **don't blow it**.
- Work hard to keep a **healthy relationship** with your **testing team**. The two teams can make each other miserable if communication goes bad.
- Recording actual time spent on a task versus estimated time on a task isn't necessary since your velocity will account for estimation errors. But, if you know something went really wrong, **it's worth discussing in the iteration review**.

Iterationcross

You've reached the end of your iteration. Take a breather and enjoy a nice crossword puzzle.



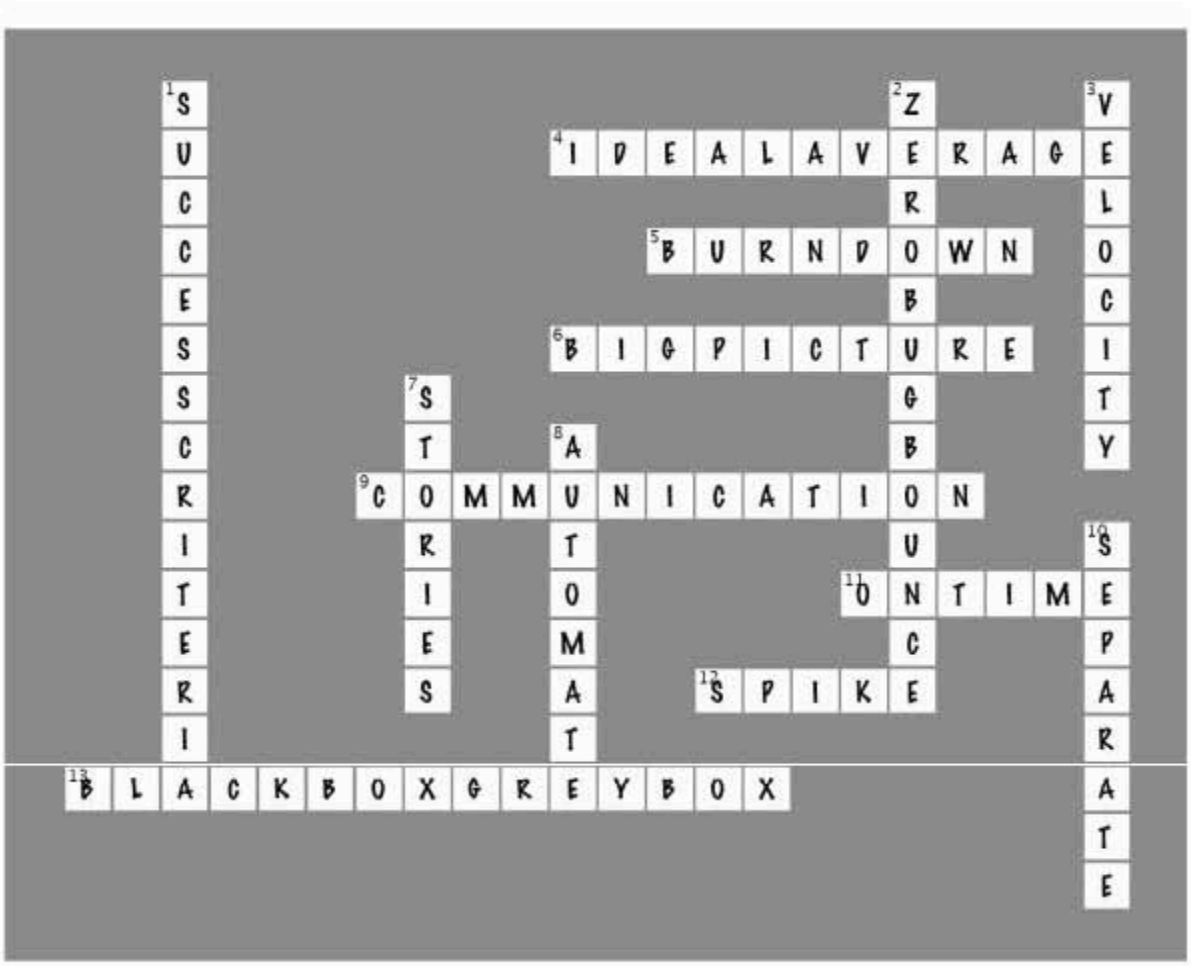
Across

4. Estimate for the ..... day and the ..... team member.
5. Pay attention to your ..... rate to help understand how your team is doing.
6. Make sure your testing team understands the ....
9. Standup meetings are about .....
11. Try really hard to end an iteration ....
12. A quick and dirty test implementation is a ..... solution.
13. System testing is usually .... testing, but sometimes .... testing.

Down

1. Since testing can usually go on forever, make sure you have this defined and agreed to by everyone.
2. When your bug fixing rate exceeds your bug finding rate for a while.
3. You should estimate consistently because random disruptions are included in your .....
7. A good way to work through a bug backlog is to treat them as .....
8. ..... system testing whenever possible.
10. System testing should really be done by a ..... team.

Iterationcross Solution



If it ain't broke... you still better fix it

So then he said, "I don't want to wear a white shirt this time; I want to wear a blue shirt." How am I supposed to react to that?

Think things are going well?

Hold on, that just might change...

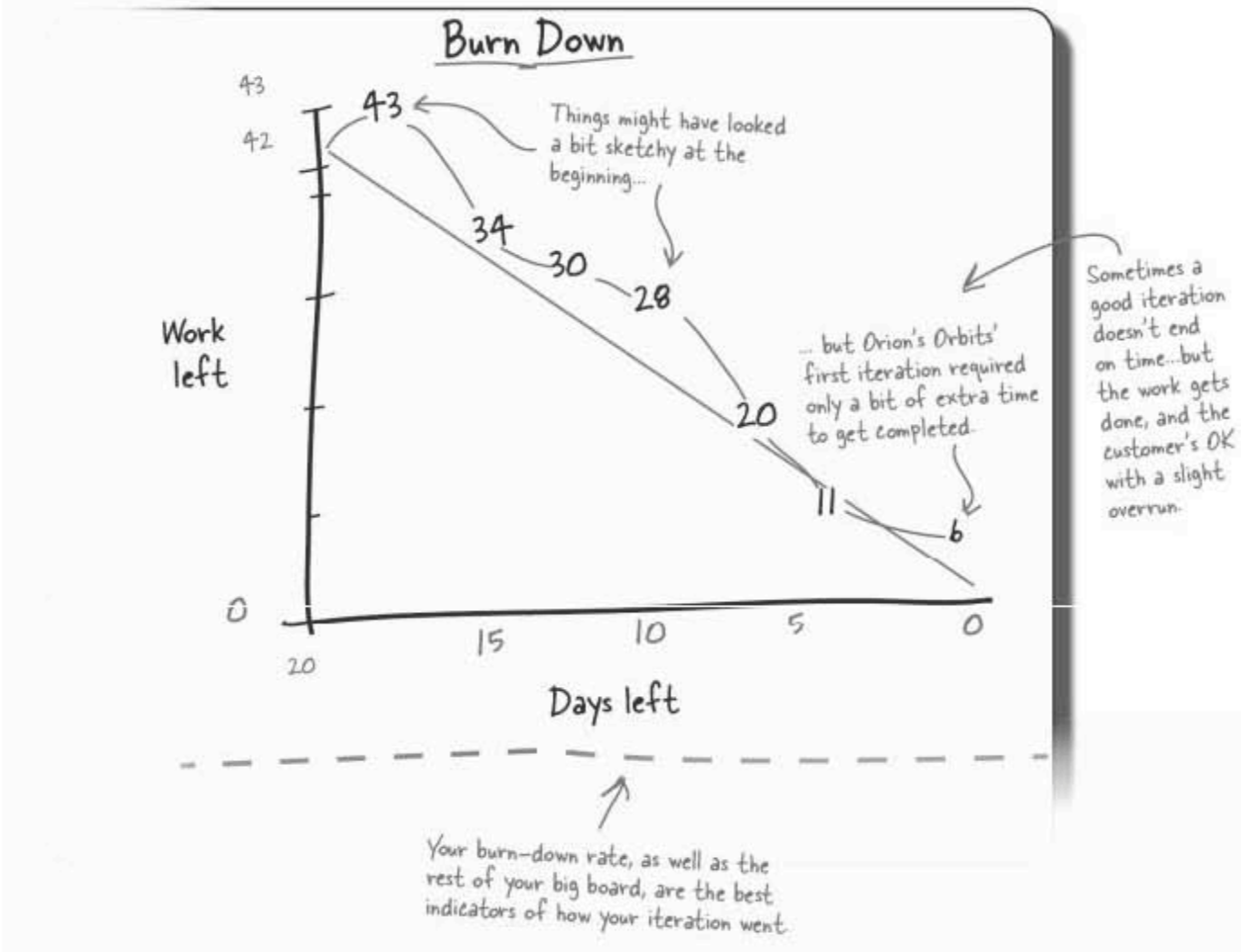
Your iteration went great, and you're delivering working software on time. Time for the next iteration? No problem, right? Unfortunately, not right at all. Software development is all about **change,** and **moving to your next iteration** is no exception. In this chapter you'll learn how to prepare for the ***next*** iteration. You've got to **rebuild your board** and **adjust your stories** and expecations based on what the customer wants ***NOW***, not a month ago.

What is working software?

When you come to the end of an iteration, you should have a buildable piece of software. But complete software is more than just code. It's also...

...completing your iteration's work

You're getting paid to get a certain amount of work done. No matter how clever your code, you've got to complete tasks to be successful.



...passing all your tests

Unit tests, system tests, black- and white-box tests...if your system doesn't pass your tests, it's not working.

```
File Edit Window PleaseWork
hfsd> java -cp junit.jar;. org.junit.runner.JUnitCore
                    headfirst.sd.chapter7.TestRemoteReader
JUn  version 4.3.1

Ti   97.825

  (112 tests)

hfsd>
```

...satisfying your customer

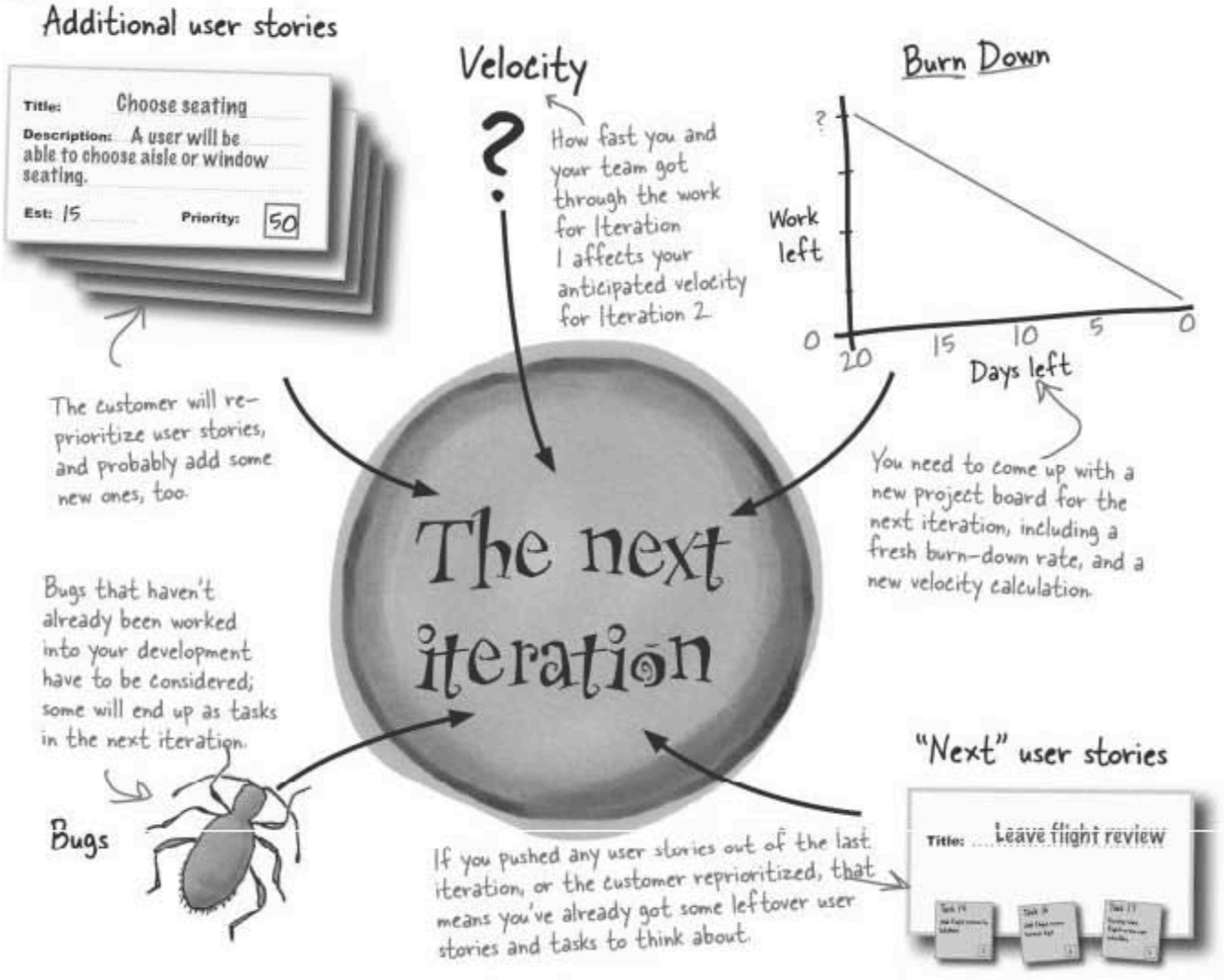
Software that does what it's supposed to do, in the time you promised it, usually makes customers pretty excited. And in lots of cases, it means you've got another iteration's worth of work.

BRAIN POWER

What would you do to get going on the next iteration?

You need to plan for the next iteration

Before diving into the next iteration, there are several things that all play a crucial part in getting ready. Here are the key things to pay attention to:



there are no Dumb Questions

Q: So what happend to the board from Iteration 1?

A: Once the iteration is finished, you can archive everything on the old Iteration 1 board. You might want to take a photo of it for archival purposes, but the important thing is to capture how much work you managed to get through in the iteration, how much work was planned, and, of course, to also take any user stories that ended up in the "Next" section back into the pack of candidate user stories for Iteration 2.

Recalculate your estimates and velocity at each iteration, applying the things you learned from the previous iteration.

You need to revise your story and task estimates and your team's velocity.

When originally planning Orion's Orbits, you and your team came up with estimates for each of the user stories for Version 1.0, and then the tasks associated with each of those user stories. Now that it's time to kick off another iteration, you have a lot more knowledge about how things will work. So it's time to revisit those estimates for the remaining user stories and tasks.

Also, when you originally calculated how much work you could complete in the first iteration, you didn't know how fast your team could develop software and complete their tasks. You probably used an initial value of something like 0.7 for your team's velocity. But that was just a rough guess...now, you and your team have completed an iteration. That means you've got hard data you can use for recalculating velocity, and getting a more accurate figure.

Remember, your estimates and your velocity are about providing confident statements to your customer about what can be done, and when it will be done. You should revise both your estimates and your velocity at every iteration.

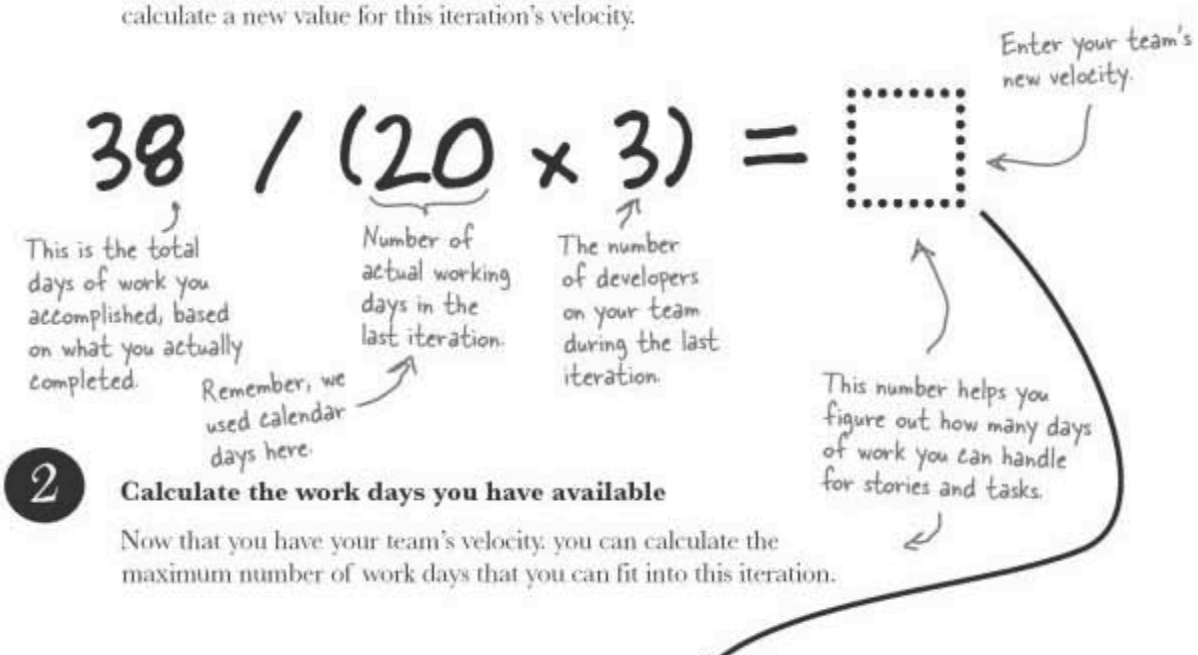
Velocity accounts for overhead in your estimates. A value of 0.7 says that you expect 30% of your team's time to be spent on other things than the actual development work. Flip back to Chapter 3 for more on velocity.

Long Exercise

It's time to plan out the work for another iteration at Orion's. First, calculate your team's new velocity according to how well everyone performed in the last iteration. Then, calculate the maximum amount of days of work you can fit into this next iteration. Finally, fill out your project board with user stories and other tasks that will fit into this next iteration using your new velocity, the time that gives you, and your customer's estimates.

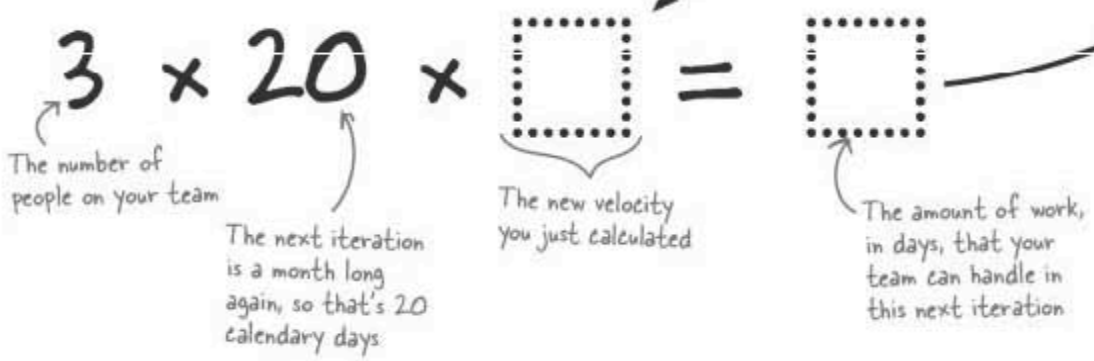
1 Calculate your new velocity

Take your team's performance from the previous iteration and calculate a new value for this iteration's velocity.



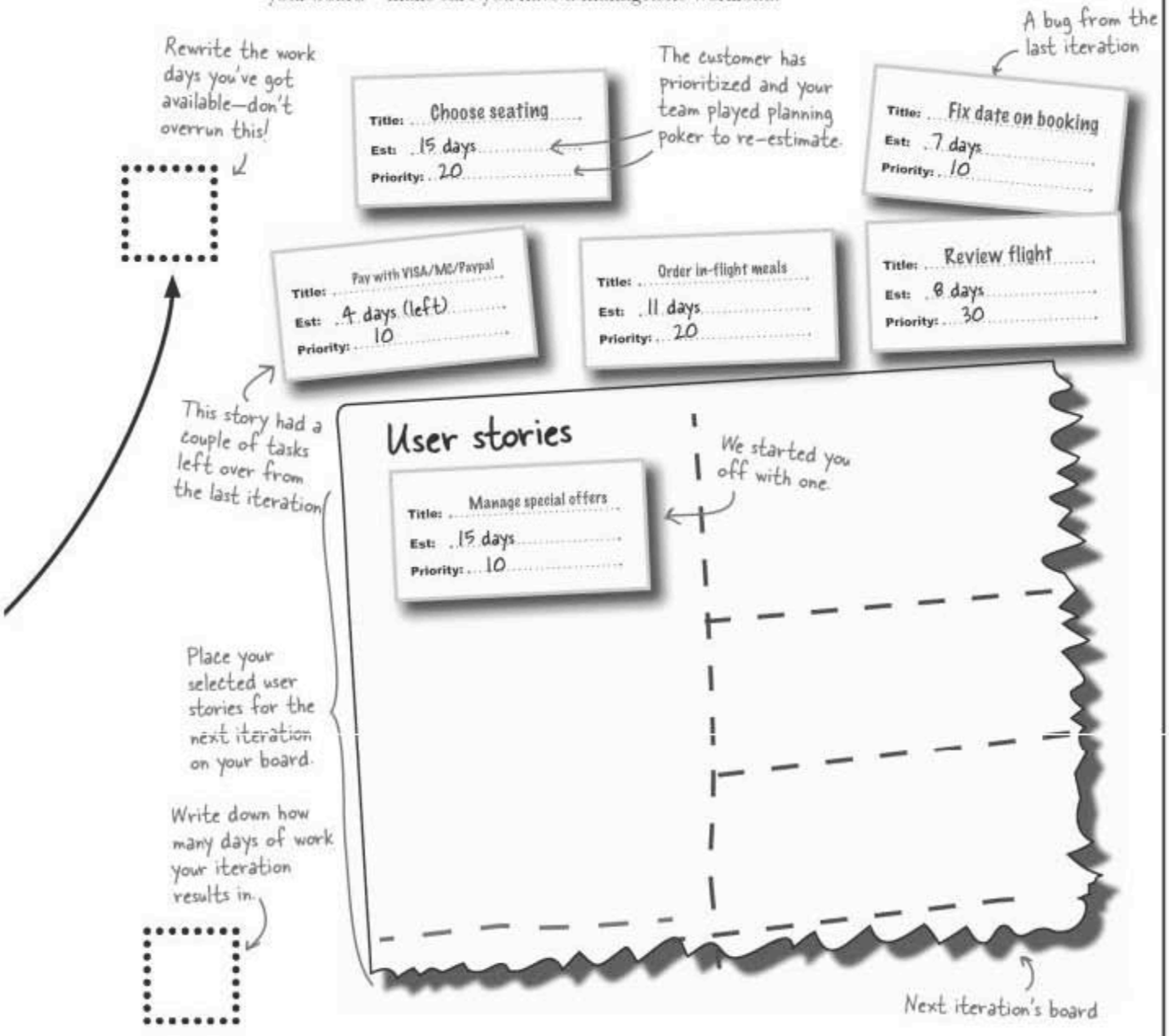
2 Calculate the work days you have available

Now that you have your team's velocity, you can calculate the maximum number of work days that you can fit into this iteration.



3 Fill up the board with new work

You know how many work days you've got, so all that's left is to take the candidate user stories and bugs, as well as stories left over the last iteration, and add them to your board—make sure you have a manageable workload.



Long Exercise Solution

Your job was to calculate your team's new velocity, the maximum amount of days of work you can fit into the next iteration, and then to fill out your project board with user stories and other tasks that will fit into this next iteration.

1 Calculate your new velocity

Take your team's performance from the previous iteration and calculate a new value for this iteration's velocity.

38 / (20 x 3) = 0.6

You got 38 days of work done, including unplanned tasks that hit the board.

Remember, velocity is a measure of how fast you and your team can get through work on your board. Regardless of whether that work is unplanned or not, it all counts.

Your team's velocity has actually dropped...

2 Calculate the work days you have available

Now that you have your team's velocity, you can calculate the maximum number of work days that you can fit into this iteration.

3 x 20 x 0.6 = 36

... as has the total amount of work that your team can execute in the next iteration.

3 Fill up the board with new work

You know how many work days you've got, so all that's left is to take the candidate user stories and bugs, as well as stories left over the last iteration, and add them to your board—make sure you have a manageable work load.

Rewrite the work days you've got available—don't overrun this!

36

Title: Choose seating
Est: 15 days
Priority: 20

These stories dropped off either because they were a lower priority or wouldn't fit within the work days left in the iteration.

Title: Order in-flight meals
Est: 11 days
Priority: 20

User stories

Title: Manage special offers
Est: 15 days
Priority: 10

Title: Fix date on booking
Est: 7 days
Priority: 10

Title: Pay with VISA/MC/PayPal
Est: 4 days (left)
Priority: 10

Title: Review flight
Est: 8 days
Priority: 30

These stories had the highest priority.

This bug was high-priority to the customer, so it was scheduled ahead of additional functionality.

This story fit within the remaining 10 work days of the iteration.

The work required for the next iteration didn't exceed the available 36 days

34

there are no Dumb Questions

Q: A team velocity of 0.6!? That's even slower than before. What happened?

A: Based on the work done in the last iteration, it turned out that your team was actually working a little *slower* than 0.7.

Q: Shouldn't my velocity get quicker as my iterations progress?

A: Not always. Remember, velocity is a measure of how fast *your* team can burn through their tasks, and 0.7 was just an original rough guess when you had nothing else to go by.

It's not uncommon for you and your team to have a tough first iteration, which will result in a lower velocity for the next iteration. But you'll probably see your velocity get better over the *next* several iterations, so you've got something to look forward to.

Q: Hmm, I noticed that some of the estimates for the Orion's Orbits user stories have changed from when we last saw them in Chapter 3. What gives?

A: Good catch! Based on the knowledge that you and your team have built up in the last iteration, you should **re-estimate** all your stories and tasks. Now you know much more about the work that will be involved so new estimates should be even more accurate, and keep you from missing something important, and taking longer than you expect.

Q: So the estimates for our user stories and their tasks will get smaller?

A: Not necessarily. They could get smaller, or bigger, but the important thing is that they will likely get more and more ***accurate*** as you progress through your iterations.

Q: I see that bug fixing is also represented as a user story. Doesn't that break the definition of a user story a bit?

A: A little, but a user story really ends up being—when it is broken into tasks—nothing more than work that you have to do. And a bug fix is certainly work for you to take on. The user story in this case is a description of the bug, and the tasks will be the work necessary to fix that bug (as far as you and your team can gauge from the description).

Q: I'm really struggling coming up with estimates for my bugs. Am I just supposed to take my best guess?

A: Unfortunately, you will be taking a best guess. And when it comes to bugs, it pays to guess conservatively. Always give yourself an amount of time that feels really comfortable to you. And remember, you've got to figure out what caused the bug as well as fix it; both steps take time.

One technique you can use is to look for similar bugs in the past and see how long they took to find and fix. That information will at least give you some guide when estimating a particular bug's work.

Q: If I have a collection of bugs, how do I decide what ones should make it into the board and be fixed in the next iteration?

A: You don't! **Priority is always set by the customer.** So the customer sets a priority for each of the bugs, and that's what tells you what to deal with in each iteration

Besides, this approach lets the customer see that for each bug that is added to the iteration, other work—like new functionality—has to be sacrificed.

The decision is functionality versus bug fixes, and it's the customer who has to make that call... because it's the customer who decides ultimately what they want delivered at the end of the ***next*** iteration.

Q: I understand why the high-priority stories made it onto the next iteration's board, but wouldn't it be a better idea to add in another high-priority user story that slightly breaks the maximum work limit, rather than schedule in a lower priority task that fits?

A: **Never** break the maximum working days that your team can execute in an iteration. That value of 36 days for the maximum amount of work your team can handle for an iteration of 20 days is exactly that: the **maximum**.

The only way that you could add more work into the iteration is to extend the iteration. You could fit in more work if your iteration were extended to, say, 22 days, but be very careful when doing this. As you saw in Chapter 1, iterations are kept small so that you can check your software with the customer often. Longer iterations mean less checks and more chance that you'll deviate further from what your customer needs.

Velocity accounts for... the REAL WORLD

Velocity is a key part of planning out your next iteration. You're looking for a value that corresponds to how fast your team **actually works**, in reality, based on how they've worked in the past.

This is why you only factor in the work that has been completed in your last iteration. Any work that got put off doesn't matter; you want to know what was done, and how long it took to get done. That's the key information that tells you what to expect in the next iteration.

Velocity tells you what your team can expect to get done in the NEXT iteration

Be confident in your estimates

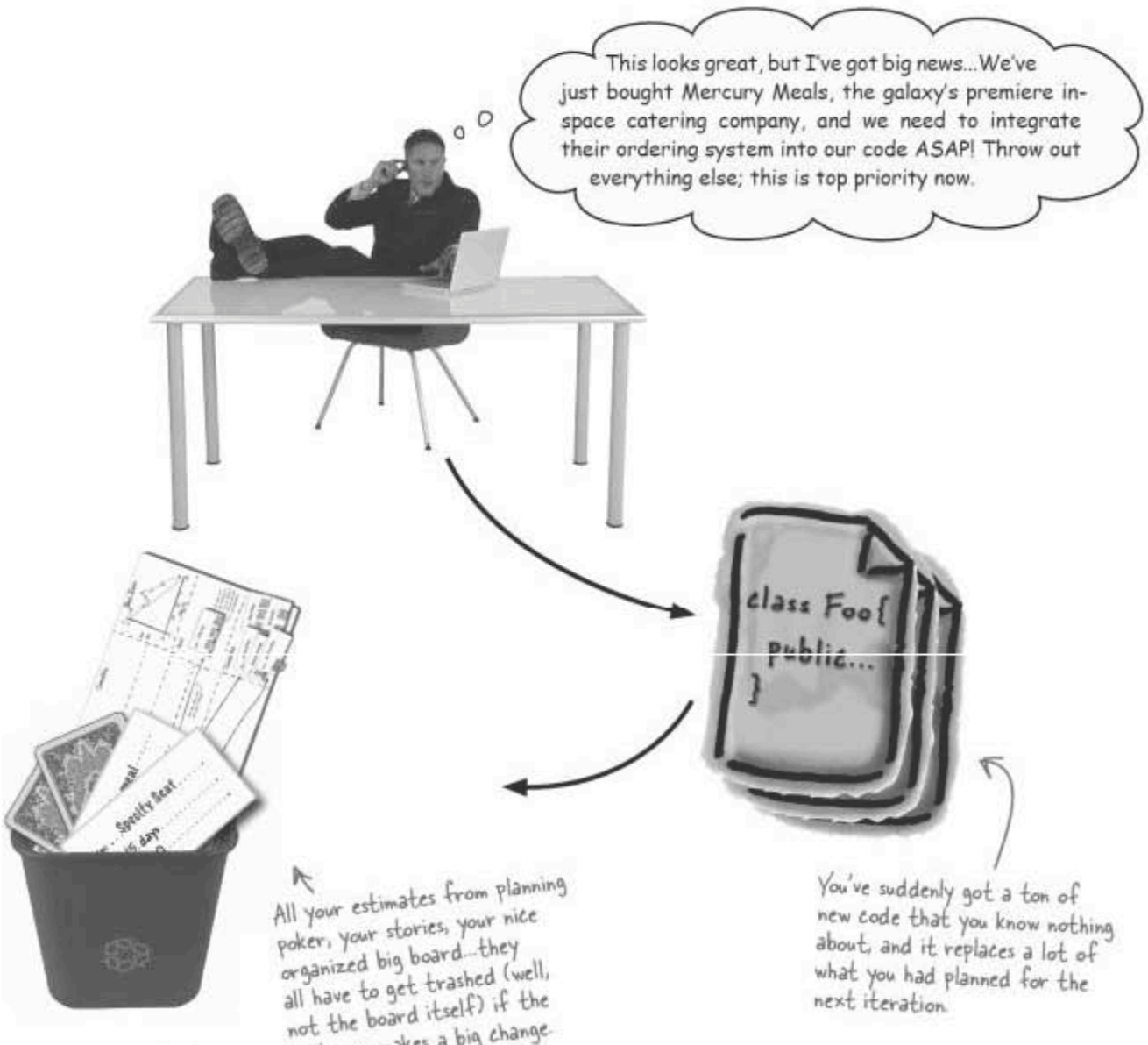
Velocity gives you an accurate way to forecast your productivity. Use it to make sure you have the right amount of work in your next iteration, and that you can successfully deliver on your promises to the customer.

By calculating velocity, you take into account the REALITY of how you and your team develop, so that you can plan your next iteration for SUCCESS.

And it's STILL about the customer

Let's say you've calculated your new velocity. You collected bugs and put them all in a bug tracker. You waded through all the piles of unfinished and delayed tasks and stories, and had the customer reprioritize those along with stories planned for the next iteration. You've got your board ready to go.

You still have to go back and get your customer's approval on your overall plan. And that's when things can go really wrong...



Software is still about CHANGE

Sometimes the customer is going to come up with a big change at the last minute. Or your best plans break down when your star programmer takes a new job. Or the company you're working for lays off an entire department...

But even though what you're working on has changed, the mechanics of planning haven't. You have a new velocity, and you know how many days of work your team has available. So you simply need to build new user stories, reprioritize, and replan.

You already know how to...

- ☐ Calculate your team's velocity
- ☐ Estimate your team's user stories and tasks
- ☐ Calculate your iteration size in days of work that your team can handle

BRAIN POWER

You've got a ton of new code that you've never seen or used before. What would be the first thing you do to try and estimate the time it will take to integrate that code into the Orion's system?

Someone else's software is STILL just software

Even though the Mercury Meals library is not code you have written, you still treat the work necessary to integrate the code into Orion's Orbits as you would any other software development activities. You'll need to tackle all the same basic activities:

User stories

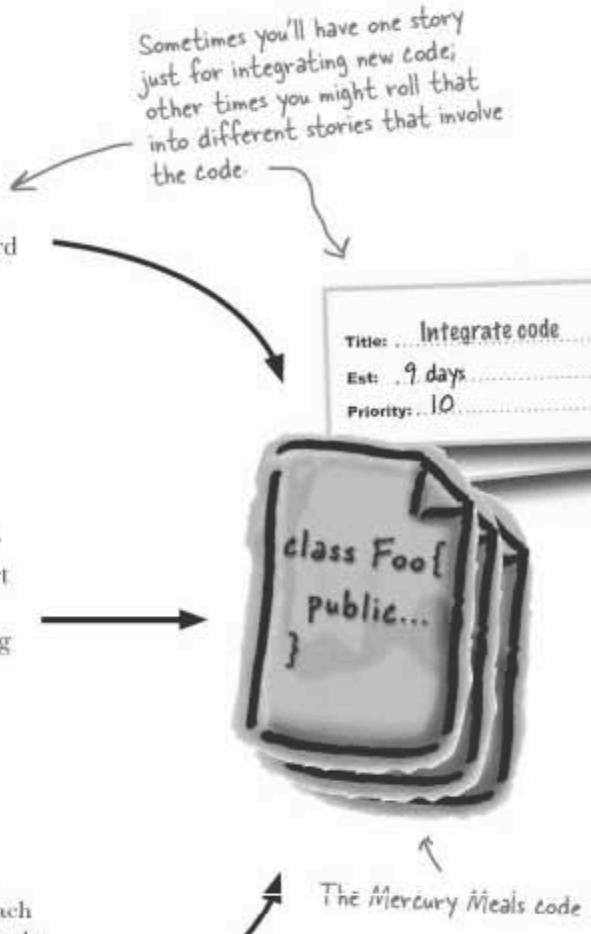
Every change to the software is motivated by and written down as a user story. In this case, your story card will be a description of how the Mercury Meals code is used by the Orion's Orbit system to achieve some particular piece of functionality.

Estimates

Every user story needs an estimate. So each of the user stories that the Mercury Meals code library plays a part in has to be estimated. How much time will it take to build that functionality, including time spent integrating the Mercury Meals code?

Priorities

The final piece of the puzzle is, of course, priorities. Each of the user stories associated with the Mercury Meals code needs to have an associated priority from your customer so that you can plan out the work for the next iteration, in the order that your customer wants it done.

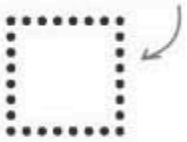


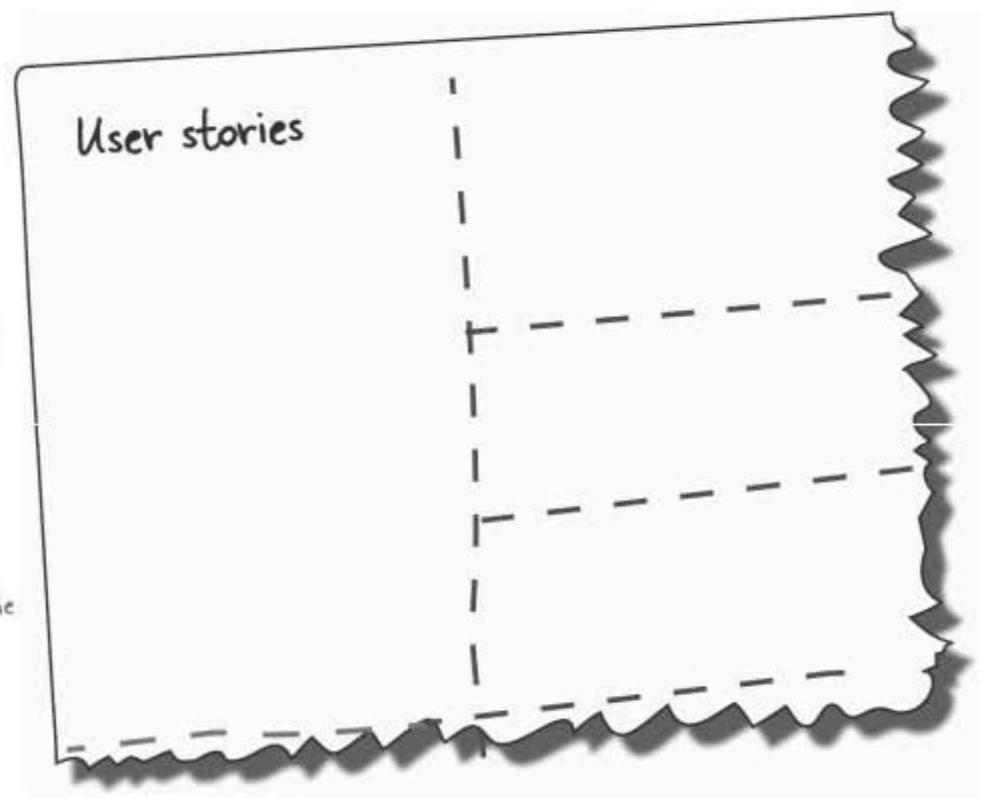
Exercise

You've got new stories related to Mercury Meals, as well as the stories you thought you'd be doing in this next iteration. Your job is to re-create the board using your velocity and the customer's priorities. (We've left out the stories that didn't make the first-pass plan.)

The number of people in your team and their velocity hasn't changed since your first attempt at a project board for this iteration, so neither has the number of work days you've got.

Add up your new total work for the next iteration.





Exercise Solution

Your job was to re-create the board using your velocity and the customer's priorities.

Title: Order vegetarian or vegan meal
Est: 6 days
Priority: 20

This Mercury Meals user story was a lower priority and so will have to wait until the following iteration.

Title: Leave flight review
Est: 8 days
Priority: 30

Title: Fix date on booking
Est: 7 days
Priority: 10

Even though it's a high priority, we couldn't fit in this bug fix so it'll be first up for the following iteration, assuming that's what the customer still wants then.

36

Your budget was 36 days of work for your team, factoring in velocity...

User stories

Title: Order Regular Meal
Est: 12 days
Priority: 10

Title: View all the orders for a flight
Est: 4 days
Priority: 10

Title: Edit Special Deals
Est: 15 days
Priority: 10

Title: Pay with Visa/MC/PayPal
Est: 4 days (left)
Priority: 10

These were the top priority in terms of the Mercury Meals user stories.

Going on priority, these two features also made it into this iteration.

... and you've planned out 35.

35

Customer approval? Check!

Once again, you've got to get customer approval once everything's planned out. And this time, there aren't any surprises...

This looks great! It's a shame we can't do everything, but I'm really excited to see us using the new Mercury Meals code. I can't wait to tell the CFO! I'll let her know right now, and I'll even mention your name.

Yes, it's really possible to get reactions like this from a customer. Good planning and giving your customer a chance for frequent feedback is a sure way to get your customer on board and excited.

there are no Dumb Questions

Q: Can you tell me again why we've got user stories for working with third-party code? And why did you estimate 12 days for ordering a meal? Isn't the whole point of getting third-party code that it saves us time?

A: A user story isn't as much about writing code as it is about what a user needs your system to do. So no matter who wrote the code, if you're responsible for functionality, capture it as a user story.
As for why the estimates are pretty large, reuse is great, but you're still going to have to write some code that interacts with the third-party software. But just think how long it would take if you had to write all the Mercury Meals code yourself.

Q: Are there any times when I shouldn't consider reusing someone else's code library or API?

A: Reuse can really give your development work a shot in the arm, but third-party code has to be used with care. When you use someone else's software, you're **relying** on that software, placing your success in the hands of the people that developed the code that *your* code now uses.

So if you're using someone else's work, you better be sure you can trust that work.

It's time to write the code for the two Mercury Meals user stories, "Order Regular Meals" and "View all the orders for a flight." On the left you have the Mercury Meals code's interface, which is a collection of methods that you can call from your own code. On the right, you need to wire up your code so that it uses the Mercury Meals API to bring both of the user stories below to life.

The two stories you have to implement

Title: Order Regular Meal
Est: 12 days
Priority: 10

Title: View all the orders for a flight
Est: 4 days
Priority: 10

Reality check: assume your team spent several days getting class diagrams together for the Mercury Meals API.

The main interface to the Mercury Meals code

How you get access to the MercuryMeals object

MercuryMeals

+ getInstance() :MercuryMeals
+ createOrder() :Order — Creates a blank order
+ submitOrder(order : Order) :boolean — Submits a completed order to Mercury Meals
+ getMealOption(name : String) :MealOption — Returns a meal option, like "Roast Beef"
+ getOrdersThatMatchKeywords(keywords : String[]) :Order[]

Methods on the interface that can be called from your code

Returns a list of orders that match a specific set of keywords

Interface that represents an order for a meal

Order

+ addMealOption(mealOption : MealOption) :void
+ addKeyword(keyword : String) :void

You can add options to an order...

...and add keywords that you can then search on later to retrieve a set of orders.

You can download the Mercury Meals code from http://www.headfirstlabs.com/books/hfsd

```
//...
// Adds a meal order to a flight
public void orderMeal(String[] options, String flightNo)
                                    throws MealOptionNotFoundException,
                                           OrderNotAcceptedException {
    MercuryMeals mercuryMeals = MercuryMeals.getInstance();

    for (int x = 0; x < options.length;x++) {

    }

    // Finds all the orders for a specific flight
    public String[] getAllOrdersForFlight(String flightNo) {

    }

// ...
```

The first line of code has already been added for you.

Add these code magnets into the main program.

```
return mercuryMeals.getOrdersThatMatchKeyword(flightNo);
```

```
Order order = mercuryMeals.createOrder();
```

```
order.addKeyword(flightNo);
```

```
MercuryMeals mercuryMeals = MercuryMeals.getInstance();
```

```
if (!mercuryMeals.submitOrder(order)) {
    throw new OrderNotAcceptedException(order);
}
```

```
MealOption mealOption = mercuryMeals.getMealOption(options[x]);
```

```
if (mealOption != null) {
    order.addMealOption(mealOption);
} else {
    throw new MealOptionNotFoundException(mealOption);
}
```

Your job was to complete the code so that it uses the Mercury Meals API to bring both of the user stories to life.

```
//...
// Adds a meal order to a flight
public void orderMeal(String[] options, String flightNo)
                                    throws MealOptionNotFoundException,
                                           OrderNotAcceptedException {
    MercuryMeals mercuryMeals = MercuryMeals.getInstance();
    Order order = mercuryMeals.createOrder();
    for (int x = 0; x < options.length;x++) {
        MealOption mealOption = mercuryMeals.getMealOption(options[x]);
        if (mealOption != null) {
            order.addMealOption(mealOption);
        } else {
            throw new MealOptionNotFoundException(mealOption);
        }
    }
    order.addKeyword(flightNo);
    if (!mercuryMeals.submitOrder(order)) {
        throw new OrderNotAcceptedException(order);
    }
}

// Finds all the orders for a specific flight
public Order[] getAllOrdersForFlight(String flightNo) {
    MercuryMeals mercuryMeals = MercuryMeals.getInstance();
    return mercuryMeals.getOrdersThatMatchKeyword((flightNo));
}

// ...
```

This gets a Mercury Meals object for this code to use

Creates a new blank order

For each of the options selected, a new option is added to the order

If an option isn't found, then an exception is raised

The flight number is added to the order as a keyword so that the orders for a particular flight can be retrieved

Attempt to submit the new complete order to Mercury Meals

Searches for and returns all orders that have the specified flight number as a keyword

there are no Dumb Questions

Q: That was easy. Why did we estimate 16 days for integrating the Mercury Meals code?

A: There's more going on here than just integrating code. First you and your team will have to come to grips with the Mercury Meals documentation. There'll be sequence diagrams to understand and class diagrams to pick through, all of which takes time. Factor in your own updates to your design and thinking about how best to integrate the code in the first place, and you've got a meaty task on your hands. In fact, it's often the thinking time up front that takes longer than the actual implementation.

Q: Does it matter if the third-party code is compiled or not?

A: If the library works then it doesn't matter if it's in source code or compiled form. You have to add in extra time to compile the code if it comes as source, but often that's an easy comand-line job and you'll have a compiled library anyway.

However, if the library doesn't work for any reason, then it really does matter if you can get at the source or not. If you are reusing a compiled library of code then you are limited to simply using that code, according to its accompanying documentation. You might be able to decompile the code, but if you're not careful, that can mean you are breaking the license of the third-party software. With compiled libraries you usually can't actually delve into the code in the library itself to fix any problems. If there's an issue, you have to try and get back in touch with the person who originally wrote the code.

However, if you are actually given the source code to the library—if it's open source or something that you've purchased—then you can get into the library itself to fix any problems. This sounds great, but bear in mind that in both cases you're trusting the third-party library to work. Otherwise you're either signing up for a barrage of questions being sent to the original developers, or for extra work to develop fixes in the code itself.

Q: What if the third-party code doesn't work?

A: Then your trust in that library quickly disappears, and you have two choices: You can continue to persevere with the library, particularly if you have the source code and can perform some serious debugging to see what is going wrong. Or you can discard the library for another, if one's available, or try to write the code yourself, if you know how.

With any of these options you are taking on extra work. That's why when you consider using third-party code, you have to think very carefully. Sometimes that code is forced upon you, like with Mercury Meals, but often you have a choice. You need to be aware of just how much trust you are putting in that library working.

Be careful when deciding to reuse something. When you reuse code, you are assuming that code WORKS.

This is a good time to go grab the code!

The code for Mercury Meals is available from the Head First Labs site. Just go to *http://www.headfirstlabs.com/books/hfsd/*, and follow the links to download the code for Chapter 10.

Testing your code

Make sure you've downloaded the Mercury Meals and Orion's Orbits code from Head First Labs. Make the additions shown on page 368, compile everything, and give things a whirl...

You should have a build tool that makes this a piece of cake.

Houston, we really do have a problem...

All that hard work has resulted in a big fat nothing. Your code...or the Mercury Meals code...***some*** code isn't working. Somewhere. And your customer, and your customer's *boss*, is about to really be upset...

Sharpen your pencil

You've just integrated a huge amount of third-party code, and something's not working. Time's short, and the pressure's on.

What would you do?

Standup meeting

Laura: We assumed that the Mercury Meals code would work, and it clearly doesn't, or at least doesn't in the way we expect. What a mess.

Bob: Well, that sounds like a reasonable assumption to me. ***We*** would never release code that doesn't work...

Mark: Yeah, but that's us. Who knows what the developers at Mercury Meals were doing?

Laura: We just took the code and assumed it would work, maybe we should have tested it out first...

Bob: So you think the developers at Mercury Meals just kicked out a dud piece of code?

Laura: It certainly looks like it. Who knows if it was ever even run, it could have been only half a project.

Mark: But it's our code and our problem now...

Bob: And it's way too late to start from scratch...

Mark: ...and we don't know how the Mercury Meals system works anyhow...

Laura: And worse than all of that, what are we going to tell the CFO? Our butts are seriously on the line here...

Trust NO ONE

When it comes to code that someone else has written, it's all about trust, and the real lesson here is to ***trust no one*** when it comes to code. Unless you've *seen* a piece of code running, or run your own tests against it, someone else's code could be a ticking time bomb that's just waiting to explode—right when you need it the most.

When you take on code from a third party, you are relying on that code to work. It's no good complaining that it doesn't work if you never tried to use it, and until you have seen it running, it's best to assume that third-party code doesn't really work at all.

Your software...your responsibility

You're responsible for how your software works. It doesn't matter if the buggy code in the software wasn't code you wrote. A bug is a bug, and as a pro software developer, you're responsible for ***all*** the software you deliver.

A third party is not you. That might sound a little obvious, but it's really important when you're tempted to assume that just because you use a great testing and development process, everyone else does, too.

Never assume that other people are following your process

Treat every line of code developed elsewhere with suspicion until you've tested it, because not everyone is as professional in their approach to software development as you are.

It doesn't matter who wrote the code. If it's in YOUR software, it's YOUR responsibility.

The Mercury Meals classes are now **your** code...but they're a mess. Circle and annotate all the problems you can see in the code below. You're looking for everything from readability of the code right through to problems with functionality.

```
// Follows the Singleton design pattern
public class MercuryMeals
{
   public MercuryMeals meallythang;
   private Order cO;
   private String qk = "select * from order-table where keywords like %1;";

public MercuryMeals() {
}

public MercuryMeals getInstance()
{
   this.meallythang = new MercuryMeals();
   return this.instance;
}
   // TODO Really should document this at some point... TBD
   public Order createOrder {
      return new Order();}

public MealOption getMealOption(String option)
throws MercuryMealsConnectionException {
   if (MM.establish().isAnyOptionsForKey(option))
   { return MM.establish.getMealOption(option).[0] };
   return null;
}
```

```
// Mercury Meals class continued...


   public boolean submitOrder(Order cO)
{
   try {
     MM mm = MM.establish();
     mm.su(this.cO);
   catch (Exception e)
   { // write out an error message } return false; }


public Order[] getOrdersThatMatchKeyword(String qk)
                                  throws MercuryMealsConnectionException {
  Order o = new Order[];
  try {
      o = MM.establish().find(qk, qk);
   } catch (Exception e) {
      return null;
   }
   return o;
}}
```

Your job was to to circle and annotate all the problems you can see in this Mercury Meals code.

```
// Follows the Singleton design pattern
public class MercuryMeals
{
   public MercuryMeals meallythang;
   private Order cO;
   private String qk = "select * from order-table where keywords like %1;";
   private MercuryMeals instance;
   public MercuryMeals() {
   }

   public MercuryMeals getInstance()
   {
      this.instance = new MercuryMeals();
      return this.instance;
   }

   // TODO Really should document this at some point... TBD
   public Order createOrder {
      return new Order();}

   public MealOption getMealOption(String option)
   throws MercuryMealsConnectionException {
      if (MM.establish().isAnyOptionsForKey(option))
      { return MM.establish().getMealOption(option).[0] };
      return null;
   }
```

No real documentation on the class, other than the fact that it tries to implement the Singleton pattern...

This attribute is public! That's a major object-oriented no-no.

Not the most descriptive of attribute names.

Why is there an Order attribute? Even a few comments would help...

Surely this should be a constant? And does qk make any sense as an attribute name?

Why have an explicit constructor declared that does nothing?

Hang on! This class is supposed to be implementing the singleton pattern but this looks like it creates a new instance of MercuryMeals every time this method is called...

Looks like the original developer just didn't ever finish this bit off.

This method seems to not do anything of any real value at the moment. You could just as easily create an order without the Mercury Meals class—and as for the indentation, it's all over the place.

Why not establish the connection just once?

Returning null is a bad practice. It's a better idea to raise an exception that gives the caller more info to work with.

```
// Mercury Meals class continued...
```

No documentation on any of this class's methods. Something that described what the methods are supposed to do would make life a LOT easier.

```
   public boolean submitOrder(Order cO)
{
   try {
     MM mm = MM.establish();
     mm.su(this.cO);
   catch (Exception e)
   { // write out an error message } return false; }
```

No wonder the software gave no indication whether it was working or not (except by just hanging...) This method swallows all exceptions that are raised. This is a classic exception anti-pattern. If an exception gets raised, and you can't deal with it locally, then pass the exception up to the caller so they can at least know what went wrong.

The code indentation is still all over the place. This makes things very hard to read.

```
public Order[] getOrdersThatMatchKeyword(String qk)
                                  throws MercuryMealsConnectionException {
   Order o = new Order[];
   try {
       o = MM.establish().find(qk, qk);
   } catch (Exception e) {
     return null;
   }
   return o;
}}
```

Which qk is being used here? This doesn't make sense, and might be a bug.

Hiding exceptions again! The caller of this method will never have to handle a MercuryMealsConnectionException or any other exception because this method is hiding anything that goes wrong and just returning null.

Believe it or not, this bracket here closes the class, but from the poor use of indentation, you'd be hard-pressed to be sure of that from looking.

You without your process

Right now things are looking pretty bleak, and without your process you would really be in trouble...

You with your process

It's not a perfect world. When your code—or someone else's code you depend on—isn't working, and your customer is breathing down your neck, it's easy to panic or catch the next flight to a non-extradition country. But that's when a good process can be your best friend.

BULLET POINTS

- When you're gearing up for the next iteration, always **check back with the customer** to make sure that the work you are planning is the work that they want done.
- You and your team's **velocity is recalculated** at the end of every iteration.
- Let your customer **reprioritize your user stories** for a new iteration, based on the working days you've got available for that iteration.
- Whether you're writing new code or reusing someone else's **it's all still just software** and your **process remains the same**.
- Every piece of code in your software, whether it be your own code or a third party's like Mercury Meals, should be represented by at least one user story.
- **Never assume anything** about code you are reusing.
- A great interface to a library of code is no guarantee that the code works. **Trust nothing** until you've seen it work for yourself.
- **Code is written once but read (by others) many times.** Treat your code as you would any other piece of work that you present to other people. It should be readable, reliable, and easy to understand.

there are no Dumb Questions

Q: Things seem to be in a really bad shape right now. What good is our process if we still end up in crappy situations like this?

A: The problem here is that when you reused Mercury Meals' software, you and your team brought in code that was developed under a different process than yours, with an entirely different result—broken code.

Not everyone developing software is going to test first, use version control and continuous integration, and track bugs. Sometimes, it's up to you to take software you didn't develop and deal with it.

Q: So how common is this situation? Couldn't I just always use my own code?

A: Most software developed today is created on really tight timelines. You have to be productive and deliver great software quicker and quicker, and often with success, so the tempo rises as your customers demand even more.

One of the best ways to save time in those situations is to reuse code—often code that your team didn't write. So the better you get at development, the more reuse will be part of your normal routine.

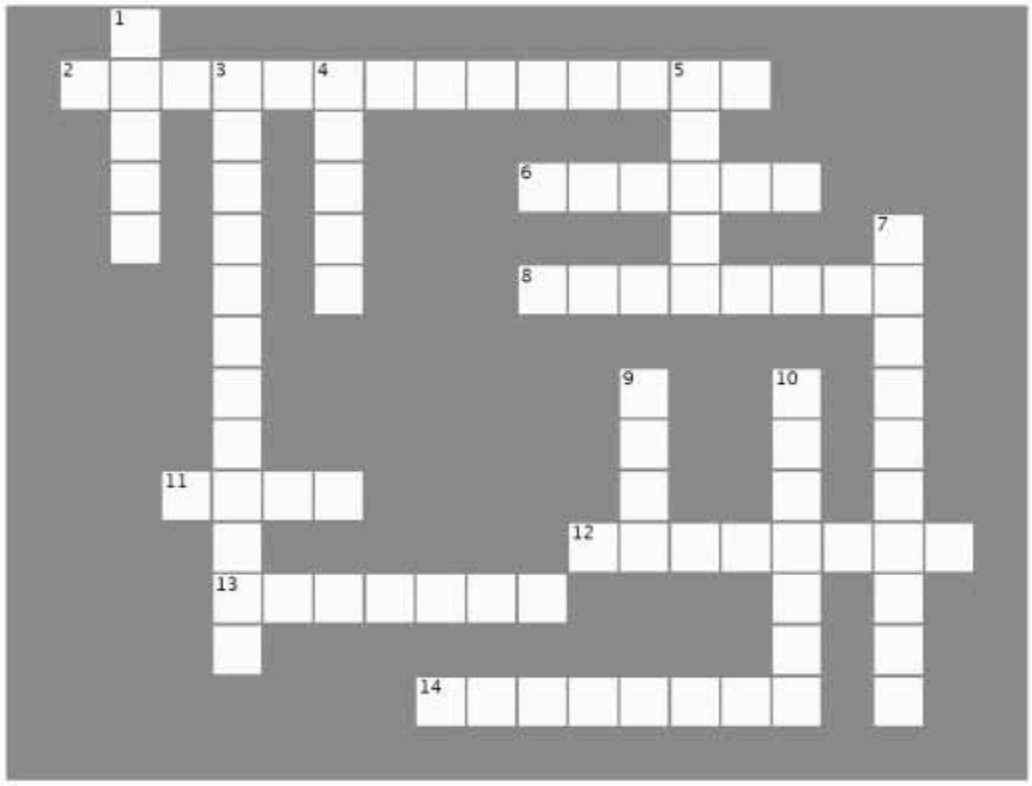
And when you start to reuse code, there's always that crucial time when you encounter code that simply does not work, and it's easier to fix that code than to start over. But hold on...Chapter 11 is all about just how to do that, without abandoning your process.

Dealing with code that doesn't work is part of software development!

In Chapter 11, you'll see how your process can handle the heat.

Software Development Cross

Let's put what you've learned to use and stretch out your left brain a bit! Good luck!



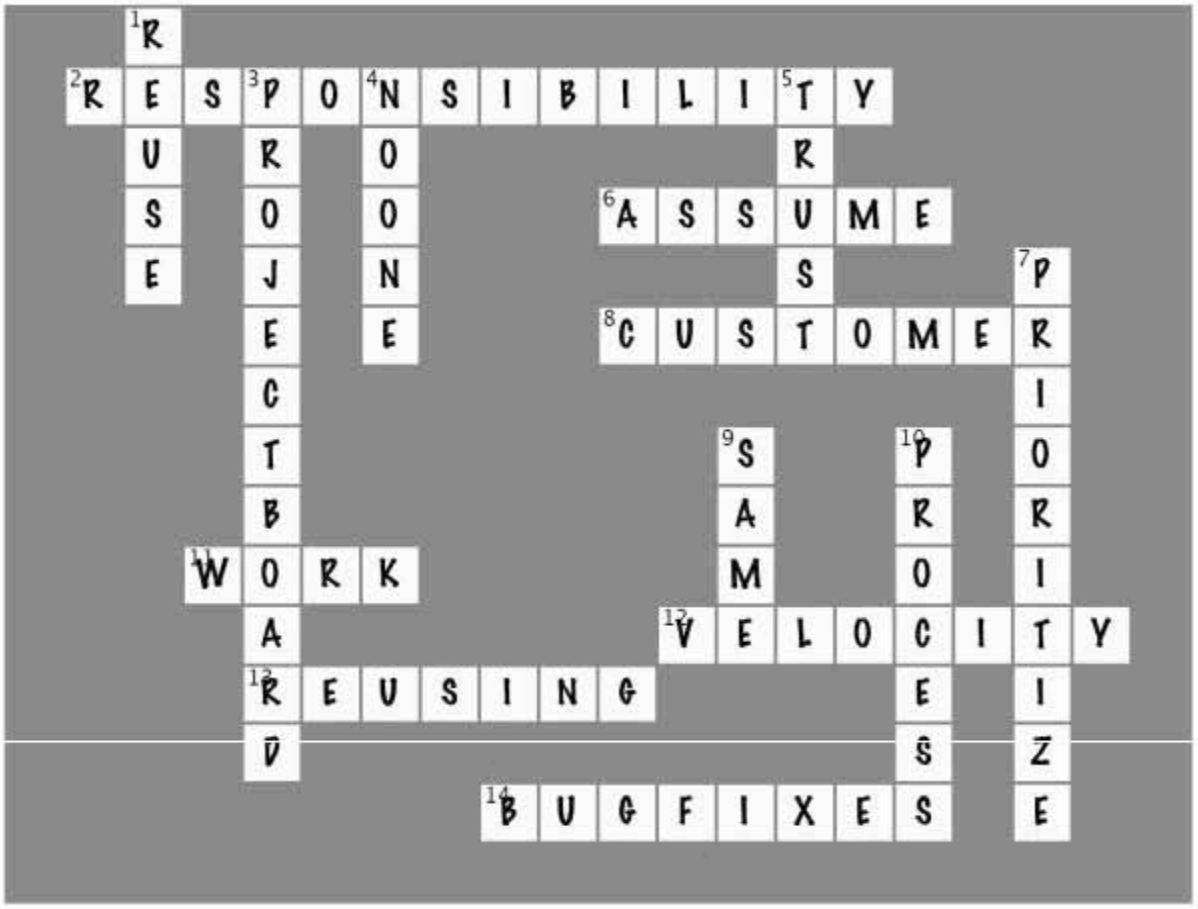
Across

2. If your software doesn't work, it's your ..... to get it fixed.
6. If you ..... that a piece of code works you are heading for a world of pain.
8. The ..... decides what is in or out for iteration 2.
11. Your velocity helps you calculate how many ..... days you can handle in iteration 2.
12. ..... deals with the real world when you're planning your next iteration.
13. Mercury Meals, other frameworks, code libraries and even code samples are all cases where you will want to consider ..... code.
14. .... are also included in the candidate work for the next iteration.

Down

1. Code ..... is one very useful technique to get you developing quickly and productively.
3. Any work for the next iteration should appear on the .... for the iteration.
4. Trust .... when it comes to reusing software.
5. Never ..... any code you haven't written or run in some way.
7. You should let your customer ..... your user stories, bug reports and other pieces of work before you begin planning iteration 2.
9. You treat third party code the ..... as your own code.
10. You may be following a great ....., but don't assume that anyone else is.

Software Development Cross Solution

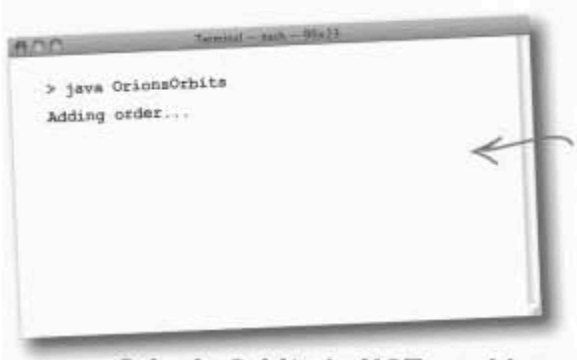


Squashing bugs like a pro

Your code, your responsibility...your bug, your reputation!

When things get tough, it's **up to you** to bring them back from the brink. **Bugs**, whether they're in your code or just in code that your software uses, are a **fact of life** in software development. And, like everything else, the way you handle bugs should fit into the rest of your process. You'll need to **prepare your board**, **keep your customer in the loop**, **confidently estimate** the work it will take to fix your bugs, and apply **refactoring** and **prefactoring** to fix and avoid bugs in the future.

At the end of the last chapter, things were in a pretty bad way. You'd added Mercury Meals' code into Orion's Orbits and were all set to demo things to the CFO when you hit a problem. Well, actually ***three*** problems—and that adds up to ***one big mess***...



Your system freezes when you try and run it...it just stops doing anything.

Orion's Orbits is NOT working.

Your customer added three new user stories that relied on some new code from Mercury Meals. Everything looked good, the board was balanced and you completed the integration work when, BOOM!, you ran your code and absolutely nothing happened. The application just froze...

You have a LOT of ugly new code.

When you dug into the Mercury Meals code, you found a ton of problems. What's causing the problems in Orion's Orbits, and where should you start looking?

You have THREE user stories that rely on your code working.

All of this would be bad enough, but there are three user stories that rely on the Mercury Meals code working, not just one.

To make matters even worse, the CEO of Orion's Orbits has talked you up to the CFO, and both are looking forward to seeing everything working, and soon...

Sharpen your pencil

Your software isn't working, you've got code to fix, and the CEO of Orion's Orbits is breathing down your neck, because the CFO is soon to be breathing down his. But how does any of this fit into your process?

What would you do next? .....

.....

.....

.....

.....

.....

.....

Wait! Think through what you would do next and fill in the blanks above before turning the page...

You, too, impatient developer... come up with a good answer, and then go on to the next section.

First, you've got to talk to the customer

Whenever something changes, talk it over with your team. If the impact is significant, in terms of functionality or schedule, then you've got to go back to the customer. And this is a big issue, so that means making a tough phone call...

Standup meeting

Laura: Well, that's why they were fired after the merger. It doesn't really matter that they screwed up, though, it's our code now...

Bob: I know, I know. But poorly written code just burns me up. It makes us look like idiots.

Mark: Look, can we move on? What do we do next?

Laura: Well, we're stuck with this code. So better start treating it like it's ours.

Mark: I think you might be on to something there...

Laura: We already know how to deal with our own new code. if we treat Mercury Meals the same way, that should at least give us a good starting point.

Bob: Ugh...you mean we have to manage its configuration, build it, and test it, don't you? Build scripts and CI all around?

Mark: Yep, we're going to have to maintain this stuff so the best first step would be to get all the Mercury Meals code into our code repository and building correctly before we can even start to fix the problem.

It's your code, so the first step is to get it building...

Broken Code Magnets

Here are a bunch of things you could do to work your way through the Mercury Meals code. Put them in the order you think you should do them. Be careful, though, there might be some you don't think will be worth doing at all.

Figure out what dependencies this code has and if it has any impact on Orion's Orbits' code.

Figure out how to package the compiled version to include in Orion's Orbits.

Organize the source code into your standard src, test, docs, etc., directories.

Put the code in your repository.

Create a place in your bug tracker for issues.

Write tests simulating how you need to use the software.

Integrate the code into your CI configuration.

Document the code.

File bugs for issues you find.

Write a build script.

Run a coverage report to see how much code you need to fix.

Get a line count of the code and estimate how long it will take to fix.

Do a security audit on the code.

Use a UML tool to reverse-engineer the code and create class diagrams.

Hint: some things may need to be done more than once.

Put your magnets on here in the order you would do them.

To-Do List

Broken Code Magnets

Here are a bunch of things you could do to work your way through the Mercury Meals code. Put them in the order you think you should do them. Be careful, though, there might be some you don't think will be worth doing at all.

To-Do List

This is a tricky one. It's going to pop up all over the place, so we put it vertically. Keep track of things as soon as you find them. You might decide it's not important later, but for now, just record everything you think could be an issue.

File bugs for issues you find.

Create a place in your bug tracker for issues.

You need to get the code building, but you'll be finding things you didn't know as you go. Set up a place to capture these right away so you don't lose this information. You'll need it to fix things later.

Organize the source code into your standard src, test, docs, etc., directories.

Keeping with the information theme, you want to capture how to build the software. In order to do that, you need a build script. It's best to get the code organized before you write your script, though, so you don't have to write it, organize the code, and then change your script.

Write a build script.

Put the code in your repository.

Next you need to get this code version controlled. By organizing it into directories before this step, you won't have to deal with shuffling around code you've already committed.

Integrate the code into your CI configuration.

Once it's in the repository, turn you CI tool loose on it. This gets your safety net set up.

Write tests simulating how you need to use the software.

You could also make a strong case for putting this in your repository before you shuffle stuff around, so you can roll things back if you mess it up.

And now we can start thinking about getting it to work. We'll talk more about this in a few pages.

So what about all the magnets we didn't use? They're not necessarily bad ideas, but here's why we didn't put them on our short list.

Figure out what dependencies this code has and if it has any impact on Orion's Orbits' code.

This is important, but we don't know what changes we're going to have to make to the code yet. We're just not ready to focus on library versions.

Figure out how to package the compiled version to include in Orion's Orbits.

This is going to be important once this code is stable, but until we get things tested and working, it's not much use worrying about how to package anything up beyond the library we already have.

Document the code.

Another important one, and it almost got a vertical spot next to "File bugs...". But since we're not making changes to the code yet, and don't even know what parts we'll need, we decided to leave this one off of our short list...although we might come back to it later.

Run a coverage report to see how much code you need to fix.

This one just can't happen yet. We don't have tests, we don't know what code we actually need, and we know some of the code isn't working. Test coverage at this point won't tell us much of value.

Get a line count of the code and estimate how long it will take to fix.

This is oh-so-tempting. It provides a solid metric to latch onto, which seems like a good thing. The problem with this is that we don't know how much of the code we'll need, and we have absolutely no idea how much is missing. What if there is a stubbed-out class where a whole section of the library is supposed to be? Concerns like this make a metric here useless.

Do a security audit on the code.

At some point this will be a great idea, but like some of the other tasks, we don't know what code we need yet, and we're about to go changing things anyway, so let's hold off on this for now.

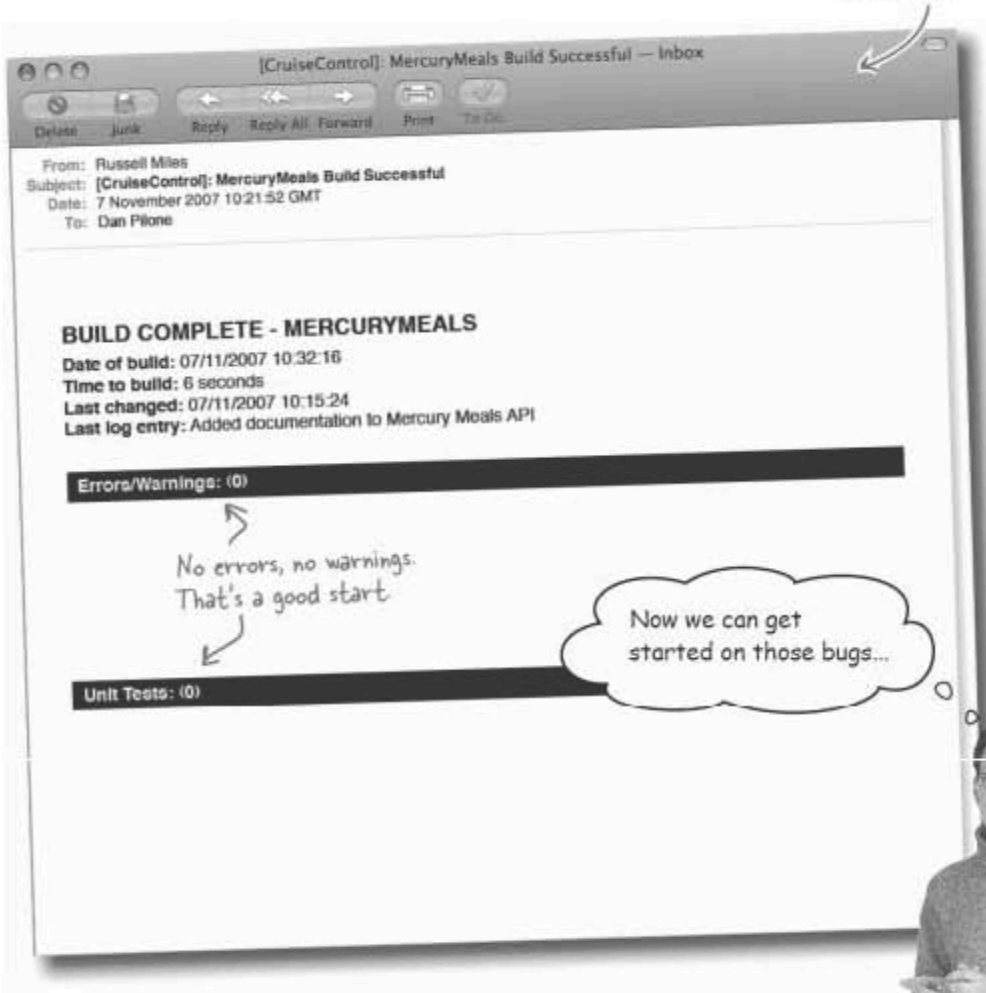
Use a UML tool to reverse-engineer the code and create class diagrams.

Of all the tasks we didn't choose to do, this is the most likely candidate to get added back in. But right now, we don't know how much of the library we need. Let's get a handle on what we have to use; then we'll try and figure out how it's supposed to work.

Priority one: get things buildable

The code is in version control, you've written build scripts, and you've added continuous integration with CruiseControl. Mercury Meals is still a junky piece of nonworking code, but at least you should have a little bit of control over the code...and that's your first priority.

An email generated by your continuous integration tool when the Mercury Meals build is run.



This assumes that you fixed anything that kept all the Mercury classes from at least compiling—but resist the urge to start fixing other things right now.

A little time now can save a LOT of time later.

None of the original bugs are fixed just yet, but that's OK. You've got a development environment set up, your code's under version control, and you can easily write tests and run them automatically. In other words, you've just prevented all the problems you've seen over the last several hundred pages from sneaking up and biting you in the ass.

You know that the code doesn't work, but now that everything is dialed into your process, you're ready to attack bugs in a sensible way. You've taken ownership of the Mercury Meals code, and anything you fix from here on out will stay fixed... saving you wasted time on the back end.

Get the code under source control and building successfully before you change anything... including fixing bugs.

We could fix code...

Now it's time to figure out what needs to be fixed. At the end of Chapter 10 you took a look at the Mercury Meals code, and the prognosis was not good...

All of these problems were found, and this was only when you peeked into the first layer of the Mercury Meals code.

```
// Mercury Meals class continued
// Follows the Singleton design pattern
public class MercuryMeals
{
   public MercuryMeals meallythang;
   private Order cO;
   private String qk = "select * from order-table where keywords like %1;";

public MercuryMeals() {
}

public static MercuryMeals getInstance()
{
   this.meallythang = new MercuryMeals();
   return this.instance;
}
   // TODO Really should document this at some point... TBD
   public Order createOrder {
      return new Order();}

public MealOption getMealOption(String option)
throws MercuryMealsConnectionException {
   if (MM.establish().isAnyOptionsForKey(option))
   { return MM.establish.getMealOption(option).[0] };
   return null;
}
```

No real documentation on the class, other than the fact that it tries to implement the Singleton pattern...

Not the most descriptive of attribute names.

Why is there an Order attribute? Even a few comments would help.

Surely this should be a constant? And does qk make any sense as an attribute name?

Why have an explicit constructor declared that does nothing?

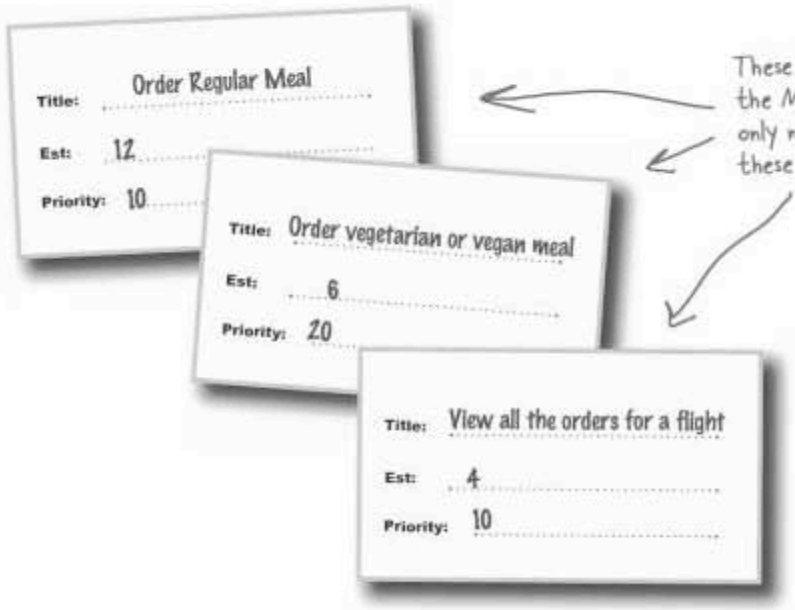
Hang on! This class is supposed to be implementing the singleton pattern, but this looks like it creates a new instance of MercuryMeals every time this method is called...

This method seems not to do anything of any real value at the moment. You could just as easily create an order without the Mercury Meals class—and as for the indentation, it's all over the place.

Returning null is a bad practice. It's a better idea to raise an exception that gives the caller more info to work with.

...but we need to fix functionality

But things might not be quite as bad as they look. You don't have to fix ***all*** the bugs in Mercury Meals; ***you just have to fix the bugs that affect the functionality that you need***. Don't worry about the rest of the code—focus just on the functionality in your user stories.



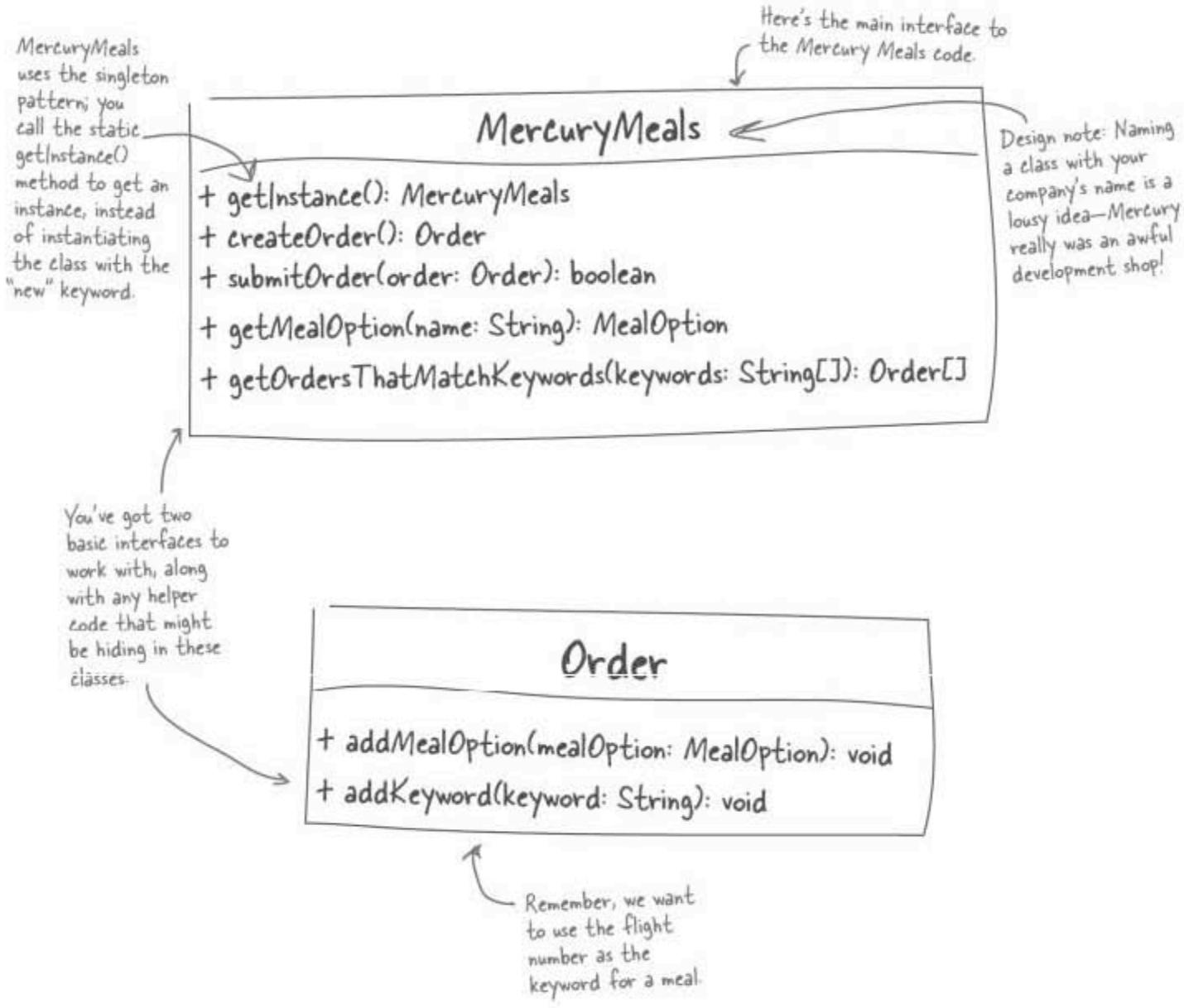
Functionality is the focus. Only fix code to fix user stories.

BULLET POINTS

- Everything revolves around **customer-oriented functionality**.
- You write and fix code to **satisfy user stories**.
- You **only fix what is broken**, and you know what is broken because you have **tests that fail**.
- **Tests are your safety net**. You use tests to make sure you didn't break anything and to know when you've fixed something.
- If there's **no test** for a piece of functionality, then it's the same as saying that functionality is broken.
- While beautiful code is great, ***functional code trumps beautiful code every single time***. This doesn't mean to let things stay sloppy, but always keep in mind why you're working on this code in the first place: **for the customer**.

Figure out what functionality works

You know that Orion's Orbits was working fine until you integrated the Mercury Meals library, so let's focus on that code. The first step is to find out what's actually working, and that means tests. Remember, if it's not testable, assume it's broken.



Your job is to create a unit test that exercises all of the functionality your user stories need. The "Order Regular Meal" test creates an order, adds a regular meal option to it (in this case, "Fish and chips"), and then submits the order to Mercury Meals. Using the class diagrams on the lefthand page, write the code for this test of the "Order Regular Meal" user story in the spaces provided below.

```
package test.com.orionsorbits.mercurymeals;
import com.orionsorbits.mercurymeals.*;
import org.junit.*;

public class TestMercuryMeals {
  String[] options;
  String flightNo;

  @Before
  public void setUp() {
    options = {"Fish and chips");
    flightNo = "VS01";
  }

  @After
  public void tearDown() {
    options = null;
    flightNo = null;
  }

  @Test
  public void testOrderRegularMeal()
    throws MealOptionNotFoundException, OrderNotAcceptedException {

    MercuryMeals mercuryMeals = MercuryMeals.getInstance();
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
    .....
  }
}
```

Title: Order Regular Meal
Est: 12 days
Priority: 10

The user story this test focuses on.

This should be a valid meal option.

The code for setUp() and tearDown() are already in place.

Throw a MealOptionNotFoundException if the meal isn't found, and an OrderNotAcceptedException if the order can't be submitted.

You may need more—or fewer—lines to implement your solution... this is what it took to get our test written.

Your job was to create a unit test that exercises all of the functionality your user stories need. The "Order Regular Meal" test creates an order, adds a regular meal option to it (in this case, "Fish and chips"), and then submits the order to Mercury Meals.

```
package test.com.orionsorbits.mercurymeals;
import com.orionsorbits.mercurymeals.*;
import org.junit.*;

public class TestMercuryMeals {
  String[] options;
  String flightNo;

  @Before
  public void setUp() {
    options = {"Fish and chips");
    flightNo = "VS01";
  }

  @After
  public void tearDown() {
    options = null;
    flightNo = null;
  }
```

Title: Order Regular Meal
Est: 12 days
Priority: 10

Here's the user story you're testing Mercury Meals functionality for.

```
  @Test
  public void testOrderRegularMeal()
    throws MealOptionNotFoundException, OrderNotAcceptedException {

    MercuryMeals mercuryMeals = MercuryMeals.getInstance();
    Order order = mercuryMeals.createOrder();
    MealOption mealOption = mercuryMeals.getMealOption(options[0]);
    if (mealOption != null) {
      order.addMealOption(mealOption);
    } else {
      throw new MealOptionNotFoundException(mealOption);
    }
    order.addKeyword(flightNo);
    if (!mercuryMeals.submitOrder(order)) {
      throw new OrderNotAcceptedException(order);
    }
  }
}
```

Even though you don't know exactly how this code works, it should be clear what it should do.

Create an order and get the single meal option that was set up prior to the test being run.

Add the "Fish and chips" meal option to the order, tie the order to the flight number, and then submit the order to Mercury Meals.

These exceptions are just ways to cause the test to fail, and say something in the Mercury Meals API didn't work.

NOW you know what's not working

Here's the build and test report email from your automated testing tool.

No errors or warnings were triggered...

...but a lot of the tests are failing.

So 13 out of 36 tests are failing...

Laura: Right, about 30% of the code we need to use is failing our tests.

Mark: But that doesn't tell us anything about how much work it will take to fix thing. And it's 30% of the code that's written...how much of that do we need?

Bob: And there could be whole chunks of code completely missing, too. I don't know how much new code we're going to have to write.

Mark: How do we estimate this?

Bob: There has to be a better way to come up with an estimate besides just guessing, right?

What would you do?

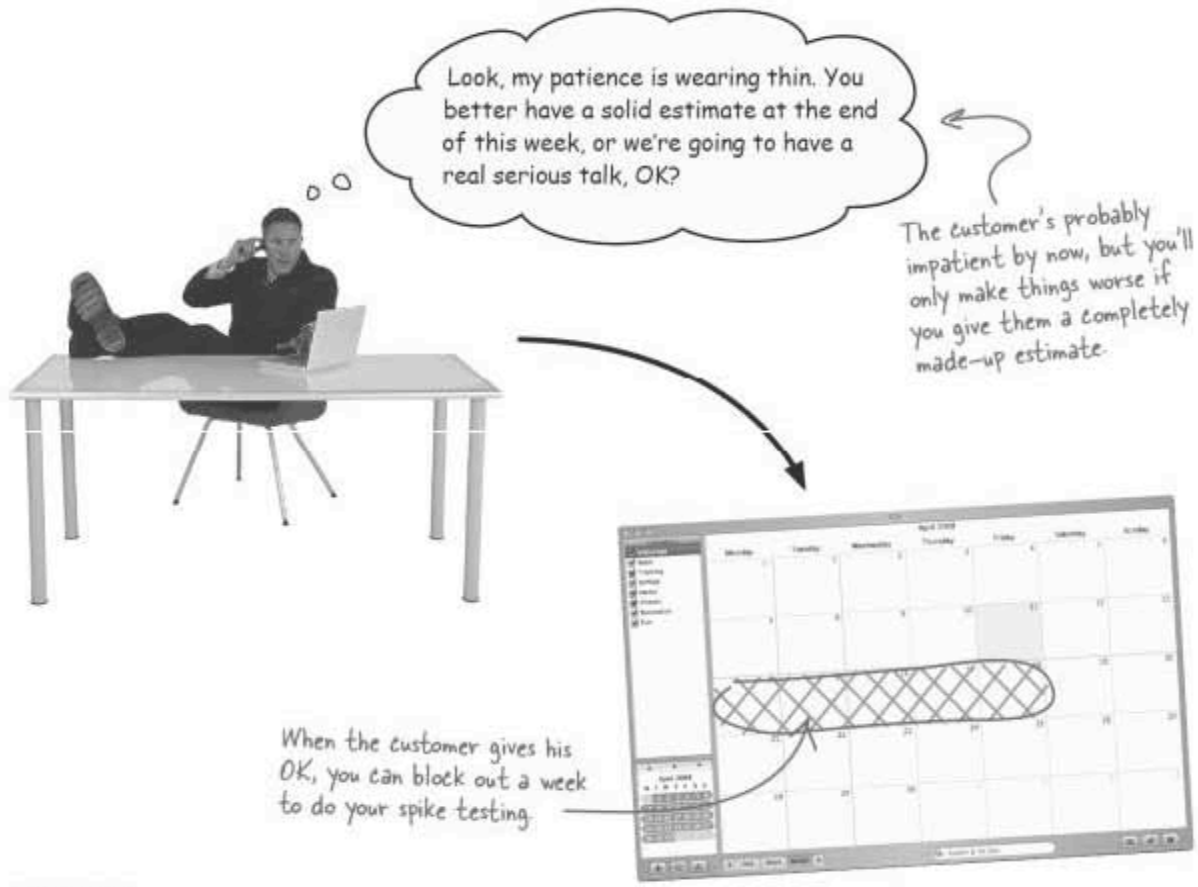
Spike test to estimate

30% of the tests you wrote are failing, but you really have no idea if a single line of code would fix most of that, or if even passing one more test could take new classes and hundreds of lines of code. There's no way to know how big a problem those 13 test failures really represent. So what if we take a little time to work on the code, see what we can get done, and then extrapolate out from that?

This is called **spike testing**: you're doing one burst of activity, seeing what you get done, and using that to estimate how much time it will take to get everything else done.

1 Take a week to conduct your spike test

Get the customer to give you five working days to work on your problem. That's not a ton of time, and at the end, you should be able to supply a reasonable estimate.



2 Pick a random sampling from the tests that are failing

Take a random sample of the tests that are failing, and try to fix just those tests. But be sure it's random—don't pick just the easy tests to fix, or the really hard ones. You want to get a real idea of the work to get things going again.

3 At the end of the week, calculate your bug fix rate

Look at how fast you and your team are knocking off bugs, and come up with a more confident estimate for how long you think it will take to fix all the bugs, based on your current fix rate.

Bugs fixed / 5 = Your daily bug fix rate

Bugs fixed: Total bugs your entire team fixed

5: Number of days in the spike test

Your daily bug fix rate: Bugs likely to be fixed per day, assuming this rate stays steady

What do the spike test results tell you?

Your tests gave you an idea as to how much of your code was failing. With the results of your spike test, you should have an idea about how long it will take to fix the remaining bugs.

The number of bugs fixed during the week-long spike test

$$4 / 5 = 0.8 \text{ bugs per day}$$

The number of work days in the spike test

Your team's bug fix rate for the spike test

You can then figure out how long it will take for your team to fix all the bugs

The bugs left, after you fixed some on the spike test

$$0.8 \times (13 - 4) = 7 \text{ days}$$

Your bug fix rate

How long it would take your whole team to fix all the remaining bugs

So we fixed some bugs AND we now know how long it will take for the entire team to fix all the problems. But I'm still not feeling very confident...

Hmm, are you sure a team will always squash bugs like they did in the spike test? How can you have any confidence that you'll fix all the bugs in seven

When it comes to bug fixing, *we really can't be sure*

When it comes down to it, a spike test really only gives you a more accurate estimate than a pure guess. It's not 100% accurate, and may not even be close.

But the spike test does give you **quantitative data** upon which you can base your estimates. You know how many bugs you fixed, and it was a random sample, so you can say with a certain degree of confidence that you should be able to fix the same number of further bugs in roughly the same amount of time.

However, a spike test does not give you any **qualitative data**. This means that we really only know how fast you can fix the bugs that we just worked on. We *don't really know how bad things might be in stuff waiting* to be fixed. There's still the potential for a bug to be in Mercury Meals that will blow your estimate out of the water, and unfortunately, that's a fact of life when it comes to bug fixing, especially on third-party software.

Your team's gut feeling matters

One quick way that you can add some qualitative feedback into your bug fix estimate is by factoring in the confidence of your team. During the spike test week, you've all have seen the Mercury Meals code, probably in some depth, so now's the time to run your fix rate past your team to factor in their confidence in that number.

Feed confidence into your estimate

Take the average of your team's confidence, in this case 70%, and factor that into your estimate to give you some wiggle room:

$$\frac{(80\% + 60\% + 70\%)}{3}$$

$$\left(0.8 \times (13 - 4)\right) \times \frac{1}{70\%}$$

The estimate for your customer

$$= 10 \text{ days}$$

to fix the remaining bugs

there are no Dumb Questions

Q: How many people should be involved in a spike test?

A: Ideally you'd get everyone that you think will be involved in the actual bug fixing involved in the spike test. This means that you not only get a more accurate estimate, because the actual people who will finish off the bug fixing will be involved in the future estimated fixing task, but those individuals also have a week to get familiar with the code.

This especially helps when you ask those members of your team to assess their confidence in the estimate that comes out of your spike test. They'll have seen the code base and have a feel for how big all the problems might have been, so their gut feeling is worth that much more.

Q: How do I pick the right tests to be part of the spike testing when I have *thousands* of tests failing?!

A: Try to pick a random sampling, but with an eye towards getting a selection of bugs that vary in difficulty. So what you're looking for first is a random sample, but then you want to make sure that you have a cross-section of bugs that, at a glance, at least appear to be challenging and not just the easiest things to fix.

Q: I thought in test-driven development, we fixed each test as we came across it. Are we not using TDD anymore?

A: This is still TDD, for sure. But this is about existing code, not writing tests for new code. You don't want to stop and try to fix each test just yet. We need a big picture view of the code right now.

However, this does cause a problem if you're using CI, and your build fails when a test fails. In that case, after you get a count of failing tests it might make sense to cheat a little and comment out the failing tests. Then add them back in one at a time. This is risky, and might get you on the TDD Most Wanted list in no time flat, but practically speaking you might want to consider it. The most important thing is you get all of those tests passing, and nothing's left commented out.

Q: Why did we add in that confidence factor again?

A: Factoring in confidence gives you that qualitative input into your estimates where your team gets a chance to say how difficult they feel the rest of the bugs may be to fix. You can take this pretty far, by playing planning poker with your bugs, but remember that the longer you spend assessing confidence, the less time you have to actually fix the bugs.

It's always a compromise between getting an absolute estimate for how long it will take to fix the bugs (and this can really only be obtained by actually fixing them all) and getting a good enough feel for how fast you can squash bugs and getting that estimate to your customer.

Q: Why five days for a spike test?

A: Good question. Five days is a good length because it focuses your team on just the spike test for a week (rather than attempting to multitask during that week), and it gives everyone enough time to do some serious bug fixing.

Q: Can I use a shorter length?

A: You can, but this will affect how many bugs your team can work through, and that affects your confidence in your final estimate. In the worst case scenario, no bugs at all are fixed in your spike test, and you're left confused and without a real end in sight.

Five days is enough time for some serious bugs to be fixed and for you to be able to come out of the spike test with some confidence in your estimate for fixing the remainder of the bugs. And in the best case scenario, you come out of the spike test week with no bugs at all!

Q: So should I do this on code we've developed, too?

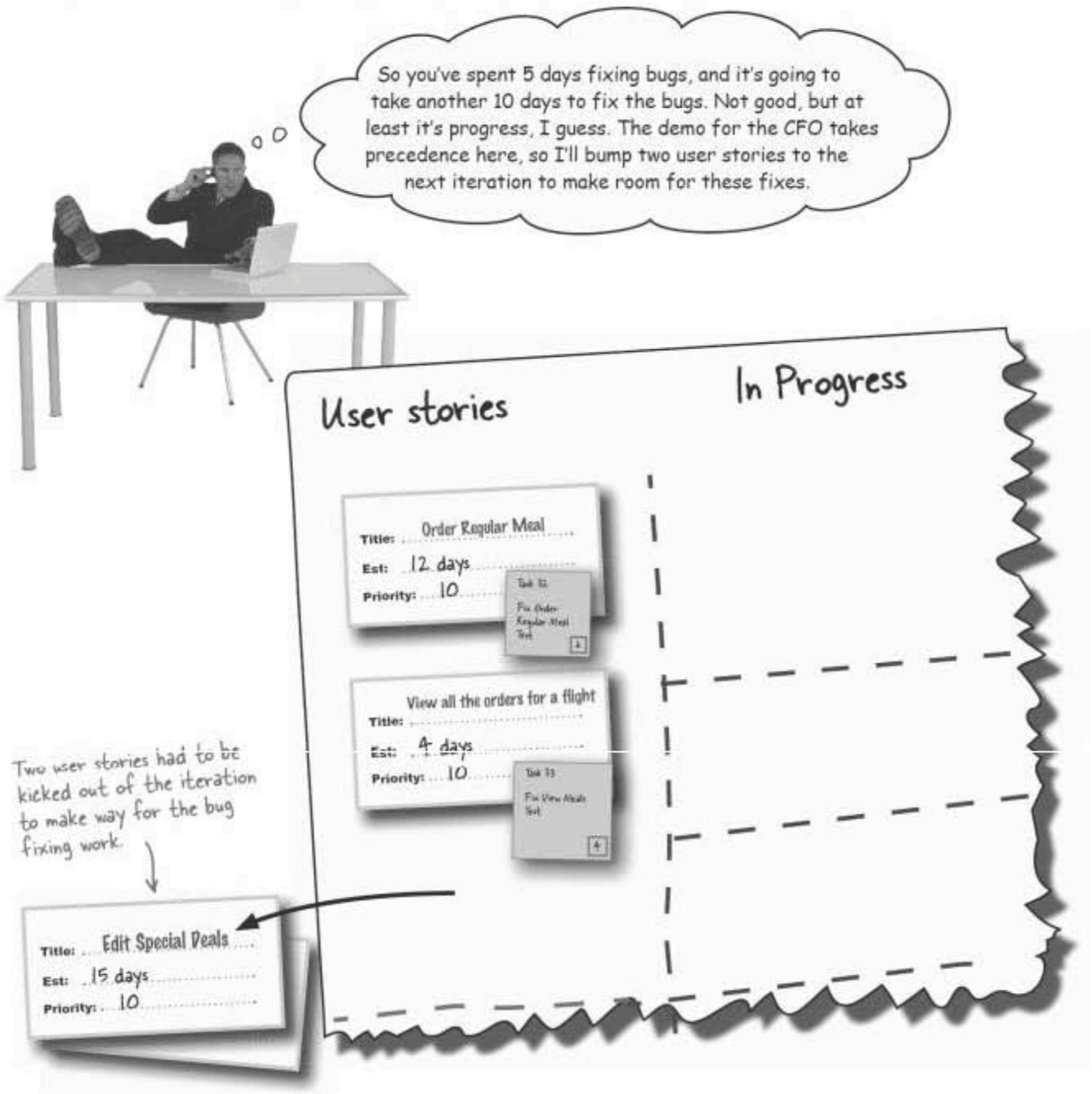
A: You really shouldn't need to. First of all, you shouldn't have a massive stack of failing tests. If a test is failing, the build should be failing, and you should fix things immediately. And with bugs, they should be prioritized in with your other work, so it's unlikely you'll suddenly get a giant stack of bugs you need to sort through. And finally, you and your team should know your code base pretty well. Your coverage reports provide value, and you know there can't be too much code involved in any given bug.

Q: How can I be absolutely sure that, even when I've factored in my team's confidence, that 10 days is definitely enough to fix all these bugs?

A: You can't. Ten days is still just an estimate, and so it's how long you think it will take, based on your spike test and your team's gut feelings. You've done everything you can to be confident in your estimate, but it is still ***just an estimate***. When it comes to bugs, you need to be aware that there is a risk that your estimates will be wrong, and that's a message that you need to convey to your customer too...

Give your customer the bug fix estimate

You've got an estimate you can be reasonably confident in, so head back to the customer. Tell him how long it will take to fix the bugs in the Mercury Meals code, and see if you can get fixing.



Exercise

Back to the magnets we didn't use on page 391. Would you do any of these activities now? Why? Any others you might add that aren't on this list?

```
Figure out what dependencies
this code has and if it has any
impact on Orion's Orbits' code.
```

Would you do this now? Why? .....
.....
.....
.....

```
Figure out how to package
 the compiled version to
include in Orion's Orbits.
```

Would you do this now? Why? .....
.....
.....
.....

```
Document the code.
```

Would you do this now? Why? .....
.....
.....
.....

```
Run a coverage report to see
how much code you need to fix.
```

Would you do this now? Why? .....
.....
.....
.....

```
Get a line count of the code and
estimate how long it will take to fix.
```

Would you do this now? Why? .....
.....
.....
.....

```
Do a security audit on the code.
```

Would you do this now? Why? .....
.....
.....
.....

```
Use a UML tool to reverse-engineer
the code and create class diagrams.
```

Would you do this now? Why? .....
.....
.....
.....

Exercise Solution

Back to the magnets we didn't use on page 391. Would you do any of these activities now? Why? Any others you might add that aren't on this list?

Figure out what dependencies this code has and if it has any impact on Orion's Orbits' code.

Would you do this now? Why? Maybe. It's possible that some kind of library conflict is behind one of our bugs. You're going to need to figure this out to get everything working by the end of the iteration anyway.

Figure out how to package the compiled version to include in Orion's Orbits.

Would you do this now? Why? Only if the current packaging approach isn't going to cut it. This is basically refactoring at the packaging level. If things are working and it's maintainable, you should probably skip this.

Document the code.

Would you do this now? Why? Absolutely! Every file you touch should come out of your cleanup with clear documentation. At a minimum, explain the code you've touched while fixing a bug.

Run a coverage report to see how much code you need to fix.

Would you do this now? Why? Probably. You now have a set of tests that scope how much of the system you need. This will give you an idea of how much of the overall code base you actually use, which is a useful metric.

Get a line count of the code and estimate how long it will take to fix.

Would you do this now? Why? Nope. Still not a terribly useful measure. Who cares how big a code base is, except as to how it relates to the functionality you need to get working?

Do a security audit on the code.

Would you do this now? Why? Yes. Any code that gets touched with your tests should be checked for security issues. If you can fix any problems as part of getting your test to pass, go for it. If not, capture it and prioritize it in a later iteration.

Use a UML tool to reverse-engineer the code and create class diagrams.

Would you do this now? Why? Maybe—it depends on how complicated the code is. If you're having trouble figuring out what a block of code is trying to do, this might help you get your head around it.

there are no Dumb Questions

Q: I noticed that the bug fixing tasks on page 406 both had estimates. Where did those estimates come from?

A: Good catch! Bug fixing tasks are just like any other type of task; they need an estimate, and there are a number of ways that you can come up with that.

You can derive the estimate, dividing the total amount of days you've calculated evenly by the number of bugs to fix, or you can play planning poker with your team. Whichever approach you take, your total planned tasks for bug fixes must never be greater than the number of days calculated from your spike test.

Q: When fixing bugs, how much time should I spend on cleaning up other problems I notice, or just generally cleaning up the code?

A: This is a tough call. It would be great to fix every bug or problem you see, but then you'll likely finish all your tasks late or, worse, end up refactoring your code indefinitely.

The best guideline is to get the code into a working, pretty decent state, within the time allotted for your bug fixing task, and then move on to the next task. First priority is to get the code working; second is to make it as easily readable and understandable as possible so that bugs are not accidentally introduced in the future. If there are problems you found but couldn't get to, file them as new bugs and prioritize them into a later iteration.

Q: What did that five-day spike test period do to our iteration length?

A: Right now, we're getting ready for the next iteration so we're between iterations. If there's a master schedule, the five days needs to be accounted for there, but in terms of iteration time, it's basically off the clock. After you get your board sorted out and everything approved by the customer, though, you should kick off a normal iteration. If you're forced to do a spike test in the middle of an iteration, that's a case where it's probably OK to slip the iteration end date by a week, assuming nearly everyone is participating.

If only a small number of developers are participating in the spike test and everyone else is continuing the iteration, you probably want to drop that five days' worth of other work from the iteration, but still end on time.

Remember, this is five days per person, not five days total.

Q: You said try and get code into a "pretty decent" state. What does that really mean?

A: This is really a judgment call, and in fact this is where you get into the aesthetics of code, which is a whole book on its own. However there are some rules of thumb that can help you decide when your code is good enough and you can move on.

First, **the code must work according to your tests**. Those tests must exercise your code thoroughly, and you should feel very confident that the code works as it should.

Secondly, **your code should be readable**. Do you have cryptic variable names? Do the lines of code read like Sanskrit? Are you using too much complicated syntax just because you can? These are all huge warning signs that your code needs to be improved in readability.

And finally, **you should be proud of your code**. When your code is correct and easily readable by another developer, then you've really done your job. It doesn't have to be perfect, but "pretty decent" starts with your code doing what it should and ends with it being readable.

Q: This sounds like the same approach as the perfect-versus-good-enough design stuff we talked about earlier, right?

A: Yes, it's based on exactly the same principle. Just as you can spend hours improving a design, trying to reach perfection, you can waste exactly the same time in your coding. Don't fall into the trap of perfection. If you achieve it, then that's great, but what you're aiming for is code that does what it should, and that can be read and understood by others. Do that, and you're coding like a pro.

Beautiful code is nice, but tested and readable code is delivered on time.

Things are looking good...

So you've picked off all the bugs from Orion's Orbits, and all functionality is working according to the results of your continuous integration build process...

Your CI tool is happy again; everything builds and passes its tests.

[CruiseControl]: MercuryMeals Build Successful — Inbox

Delete Junk Reply Reply All Forward Print To Do

From: Russell Miles
Subject: [CruiseControl]: MercuryMeals Build Successful
Date: 7 November 2007 10:21:52 GMT
To: Dan Pilone

BUILD COMPLETE - MERCURYMEALS

Date of build: 07/11/2007 10:32:16
Time to build: 6 seconds
Last changed: 07/11/2007 10:15:24
Last log entry: Added documentation to Mercury Meals API

Errors/Warnings: (0)

No errors...

Unit Tests: (36/36)

All tests passed

... and all the tests are passing!

File Edit Window Help Spike!

```
hfsd> java -cp junit.jar;. org.junit.runner.JUnitCore
                    test.com.orionsoribits.mercurymeals.TestMercuryMeals
JUnit version 4.3.1
.
Time: 75.999
OK (36 tests)

hfsd>
```

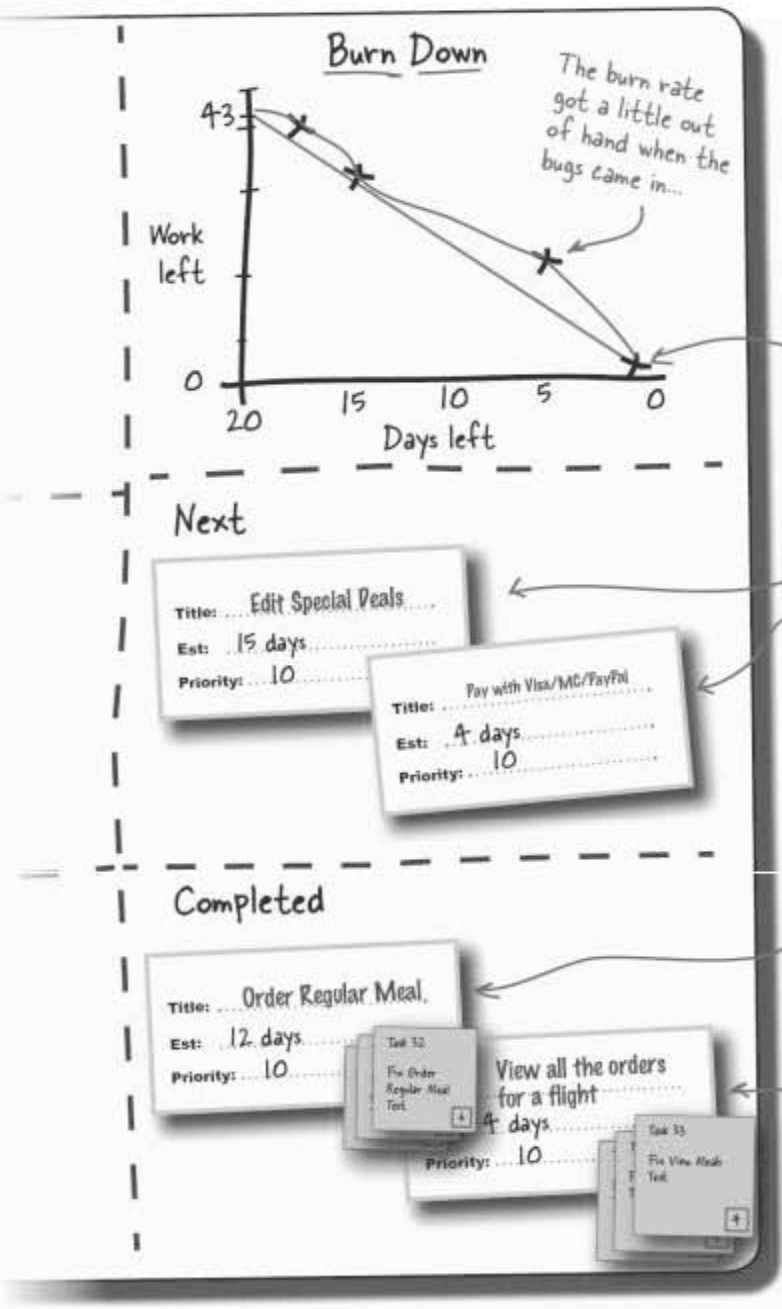
The direct output from testing the new Mercury Meals functionality

Here's the most important part. OK. No failing tests!

...and you finish the iteration successfully!

Remember, success changes as your iteration goes on. In this case, success turned out to mean dropping two stories, but getting the CFO demo done.

You've reached the end of this iteration and, by managing the work and keeping the customer involved, you've successfully overcome the Mercury Meals bug nightmare. Most importantly, you've developed what your customer needed.



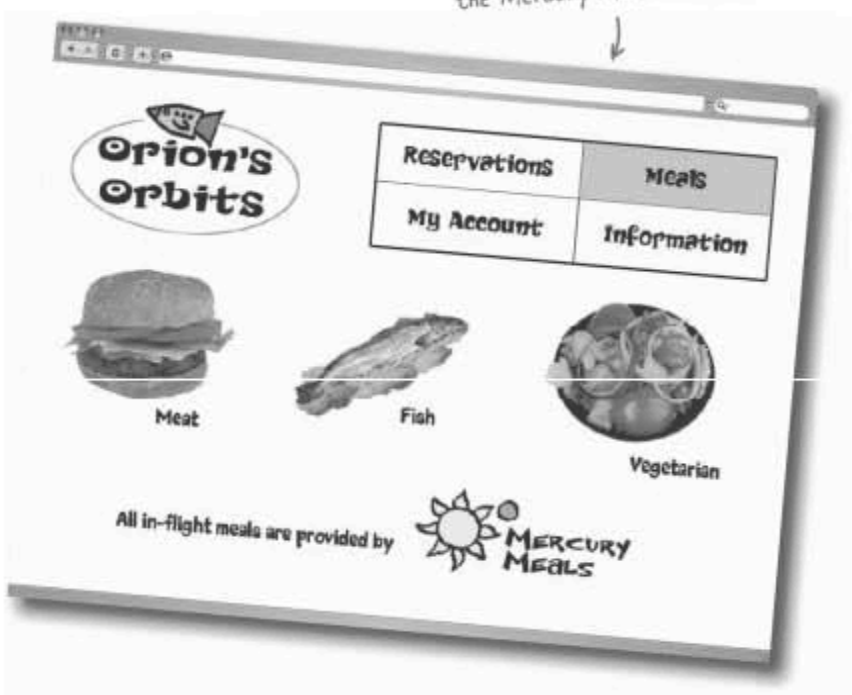
AND most importantly the customer is happy

You and your team of developers, by applying your best practices and professional process, have overcome the perils of integrating third-party code, fixed the bugs that arose from that integration, and have delivered the demo on time. The CFO, who just cares that things work, is pretty stoked.

You got the Mercury Meals integration working!? That's fantastic, guys; you really deliver! You'll all be featured in my review to the board—better start ordering those PlayStation 3's now!

The Orion's Orbits CFO...you know, the woman who signs your paychecks (and approves raises)

The all-new Orion's Orbits, with the Mercury Meals features



But wait a sec, isn't there a lot of code in Mercury's Meals that we haven't tested? We've only proven the parts of the Mercury Meals code that are used by our user stories, but doesn't that mean you're shipping software that could contain a stack of buggy code? That can't be right, can it?

You've uncovered an unfortunate truth.

Yes, there may be bugs in the code, particularly in the Mercury Meals code that you inherited. ***But you delivered code that worked.***

Yes, there are potentially large pieces of that library that haven't yet been covered by tests. ***But you have tested all the code that you actually use to complete your user stories.***

The bottom line is that pretty much *all software has some bugs.* However, by applying your process you can avoid those bugs rearing their ugly head in your software's functionality.

Remember, your code doesn't have to be perfect, and often good enough is exactly that: good enough. But as long as any problems in the code don't result in bugs (or software bloat), and you deliver the functionality that your customer needs, then you'll be a success, and get paid, every time.

Security issues are the one exception. You need to be careful that code that isn't tested isn't available for people to use—either accidentally or deliberately. Your coverage report can help identify which code you're actually using.

Real success is about DELIVERING FUNCTIONALITY, period.

Tools for your Software Development Toolbox

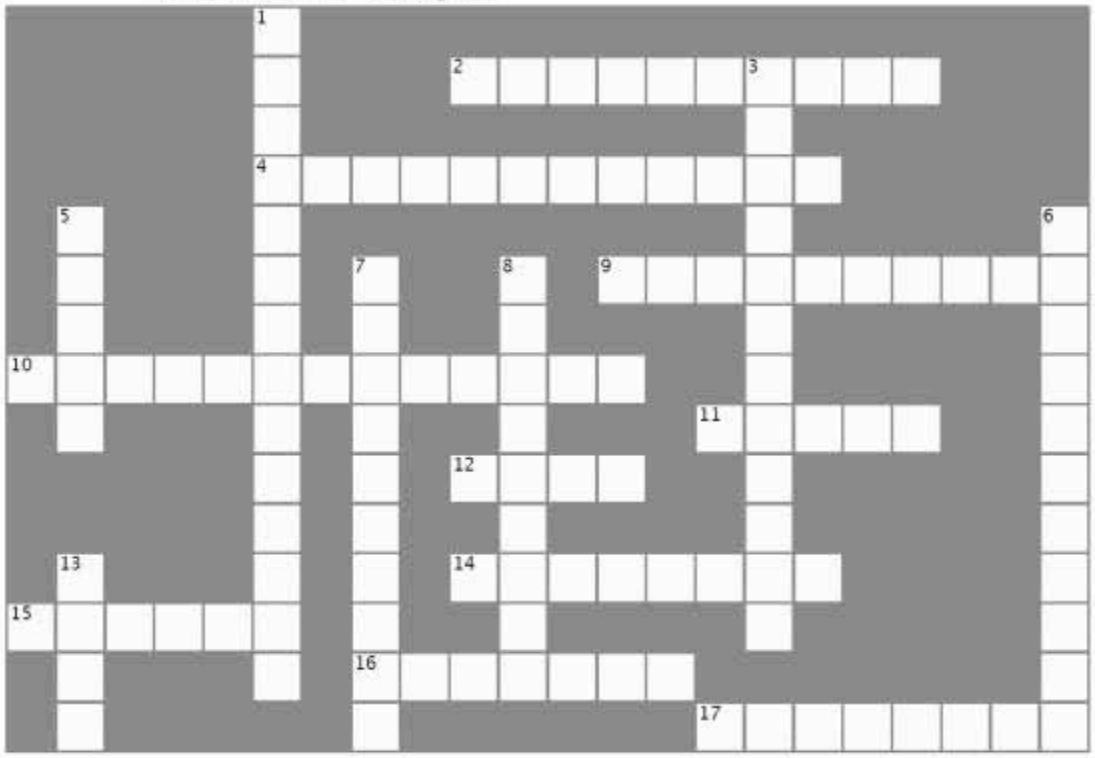
Software Development is all about developing and delivering great software. In this chapter, you learned how to debug like a pro. For a complete list of tools in the book, see Appendix ii.

BULLET POINTS

- Before you change a single line of code, **take ownership** of it by adding it into your build process and putting it under source code management.
- **Take responsibility** for all the code in your software. If you see a problem, then don't cry "it's someone else's code"; write a test, then **fix it**.
- **Don't assume** a single line of **code works** until there is a **test** that proves it.
- **Working software** comes first; **beautiful code** is second.
- Use the **pride test**. If you'd be happy for someone else to read your code and rely on your software, then it's probably in good shape.

BugSquashingCross

Flex your brain with this crossword puzzle. All of the words below are somewhere in this chapter.



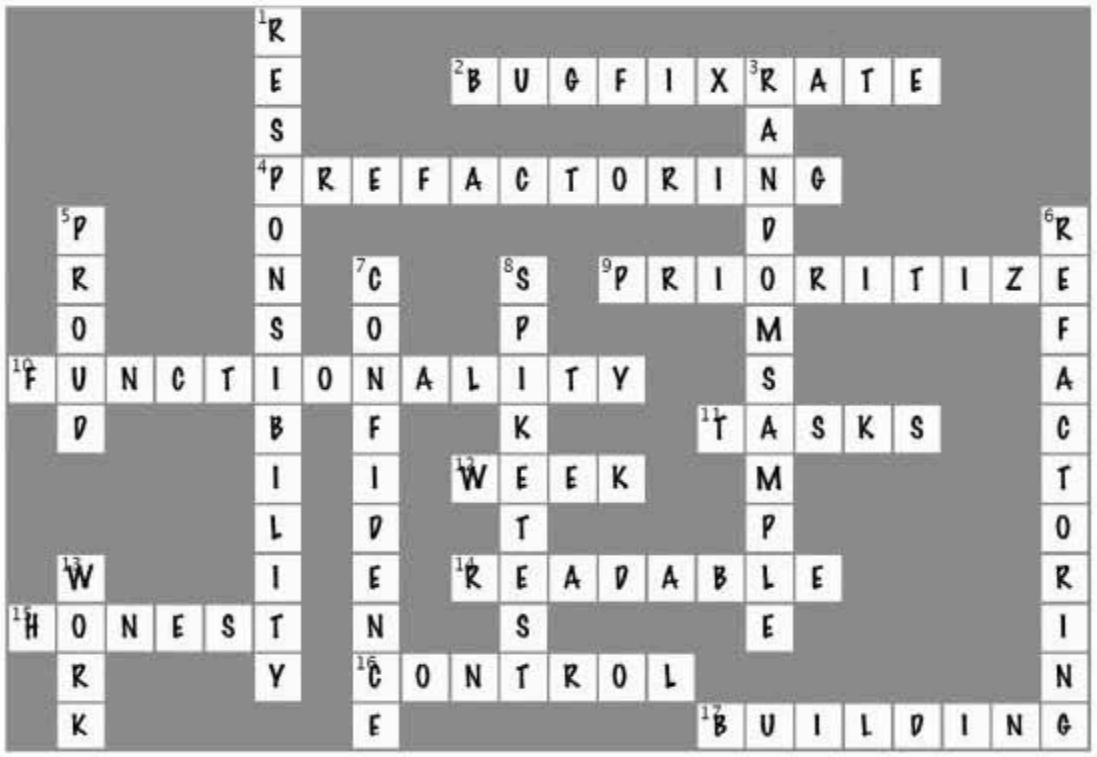
Across

2. At the end of a spike test you have a good idea what your team's .... is
4. When you apply your refactoring experience to avoid problems up front, that is called ....
9. When new bug fix tasks appear on your board, your customer might need to re-.... the work left in the current iteration.
10. When fixing bugs you are fixing ....
11. Fixing bugs becomes .... or sometimes full stories on your board.
12. A spike test should be around a .... in duration.
14. Close second priority is for your code to be .... and understandable by other developers
15. You should always be .... with your customer
16. The first step when dealing with a new chunk of unfamiliar code is to get it under source code ....
17. Before you change anything, get all your code ....

Down

1. Take .... for all the code in your software, not just the bits that you wrote
3. The best spike tests include attempting to fix a .... of the bugs.
5. You should be .... of your software
6. When you change code to make it work or just to tidy it up, this is called ....
7. You can account for your team's gut feeling about a collection of bugs by factoring in their .... in your big fixing estimate.
8. To help you estimate how long it will take to fix a collection of bugs in software you are unfamiliar with, use a ....
13. Top priority is for your code to ....

BugSquashingCross Solution



Having a process in life

You've learned a lot about software development. But before you go pinning burn-down graphs in everyone's office, there's just a little more you need to know about dealing with each project—on its own terms. There are a lot of **similarities** and **best practices** you should carry from project to project, but there are **unique** things everywhere you go, and you need to be ready for them. It's time to look at how to apply what you've learned to **your particular project**, and where to go next for **more learning**.

Pinning down a software development process

You've read a lot of pages about software development process, but we haven't pinned down exactly what that term really means.

A **software development process** is a structure imposed on the development of a software product.

Wikipedia's definition of a software development process.

Notice that definition doesn't say "a software development process is four-week iterations with requirements written on index cards from a user-focused point of view..." ***A software development process is a framework that should enable you to make quality software.***

There is no silver-bullet process

There's no single process that magically makes software development succeed. A good software process is one that lets ***your*** development team be successful. However, there are some common traits among processes that work:

- ☐ **Develop iteratively**. Project after project and process after process have shown that big-bang deliveries and waterfall processes are extremely risky and prone to failure. Whatever process you settle on, make sure it involves developing in iterations.

- ☐ **Always evaluate and assess**. No process is going to be perfect from day one. Even if your process is really, really good, your project will change as you work on it. People will be promoted or quit, new developers will join the team, requirements will change. Be sure to incorporate some way of evaluating how well your process is working, and be willing to change parts of the process where it makes sense.

- ☐ **Incorporate best practices**. Don't do something just because it's trendy, but don't avoid something because it's trendy either. Most of the things that people takes for granted as good software development started out as a goofy idea at some point. Be critical—but fair—about other processes' approaches to problems, and incorporate those approaches when they might help your project. Some people call this ***process***

A great software process is a process that lets YOUR development team be successful.

A good process delivers good software

Let's say your team loves its process. But suppose your team has yet to deliver a project on time, or deliver software that's working correctly. If that's the case, you may have a **process problem**. The ultimate measure of a process is how good the software is that the process produces. So you and your team might need to change a few things around.

Before you go changing things, you need to be careful—there are lots of wrong ways to change things. Here are a few rules to think about if you're considering changing part (or even all) of your process:

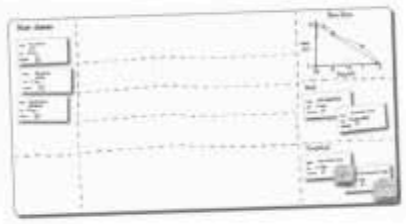
1. **Unless someone is on fire, don't change things mid-iteration.**
Changes are usually disruptive to a project, no matter how well-planned they are. It's up to you to minimize disruptions to other developers. Iterations give you a very natural breaking point. And good iterations are short, so if you need to change your process, ***wait until the end of your current iteration***.

2. **Develop metrics to determine if your changes are helping.**
If you're going to change something, you'd better have a good reason. And you should also have a way to measure whether or not your change worked. This means every change is examined at least twice: first, to decide to make the change, and then again—at least an iteration later—to measure if the change was a good idea or not. Try to avoid touchy-feely measures of success, too. Look at things like test coverage, bug counts, velocity, standup meeting durations. If you're getting better numbers and better results, you've made a good change. If not, wait for the next iteration, and be willing to change again.

3. **Value the other members of your team.**
The single biggest determinant of success or failure on a project are the people on your team. No process can overcome bad people, but good people can sometimes overcome a bad process. Respect your fellow team members—and their opinions—when evaluating your process and any changes you might want to make. This doesn't necessarily mean you have to run everything by committee, but it does mean you should try and build consensus whenever possible.

If you could change one thing about your current software process, what would it be? Why? How would you measure whether or not your change was effective?

Below are some of the best practices you've learned about in earlier chapters. For each technique, write down what you think it offers to a software process, and then how you could measure whether or not that technique helped ***your*** project.



The big board

What does this technique offer? .....

.....

.....

How do you know if it worked? .....

.....

.....

User stories

What does this technique offer? .....

.....

.....

How do you know if it worked? .....

.....

.....

Version control

What does this technique offer? .....

.....

.....

How do you know if it worked? .....

.....

.....

Continuous integration (CI)

What does this technique offer? .....

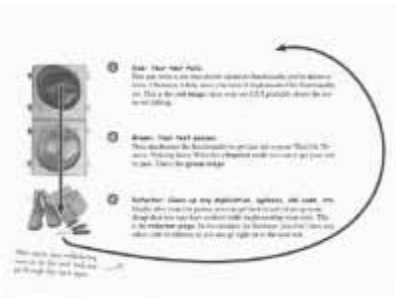
.....

.....

How do you know if it worked? .....

.....

.....



Test-driven development (TDD)

What does this technique offer? .....

.....

.....

How do you know if it worked? .....

.....

.....

Test coverage

What does this technique offer? .....

.....

.....

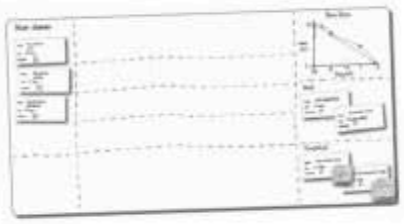
How do you know if it worked? .....

.....

.....

Exercise Solution

Below are some of the best practices you've learned about in earlier chapters. For each technique, you were asked to write down what you think each technique offers, and then how you could measure whether or not that technique helped ***your*** project.



The big board

What does this technique offer? Everyone on the team knows where they are, what else needs to be done, and what has to happen in this iteration. You can also see if you're on schedule.

How do you know if it worked? There should be fewer bugs resulting from missed features, better handling of unplanned items, and an idea of exactly what's done during this iteration.

User stories

What does this technique offer? A way to split up software requirements, track those requirements, and make sure the functionality the customer wants is captured correctly.

How do you know if it worked? There should be fewer misunderstandings about functionality. Velocity on a project should also go up, since developers know what to build better.

Version control

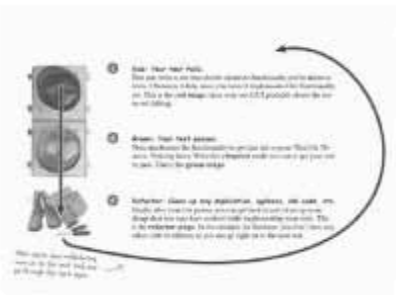
What does this technique offer? Changes can be distributed across a team without risking file loss and overwrites. You can also tag and branch and keep up with multiple versions.

How do you know if it worked? No code overwrites, no code lost from bad merges, and changes to one part of software shouldn't affect other pieces and cause them to break.

Continuous integration (CI)

What does this technique offer? The repository always builds because compilation and testing are part of check-in, and the code in the repository always works.

How do you know if it worked? Nobody checks out code and finds out it doesn't work, or doesn't compile. Bug reports should go down, since code must pass tests to be checked in.



Test-driven development (TDD)

What does this technique offer? A way to ensure your code is testable from the very beginning of development. Also introduces test-friendly patterns into your code.

How do you know if it worked? Fewer bugs because testing starts earlier. Better coverage, and every line of code matters. Possibly better design, and less legacy code.

Test coverage

What does this technique offer? Better metrics on how much code is being tested and used. A way to find bugs because they usually exist in untested and uncovered code.

How do you know if it worked? Bugs becomes focused on edge cases because the main parts of code are well-tested. Less unused or "cruft" code that's uncovered and not useful.

Formal attire required...

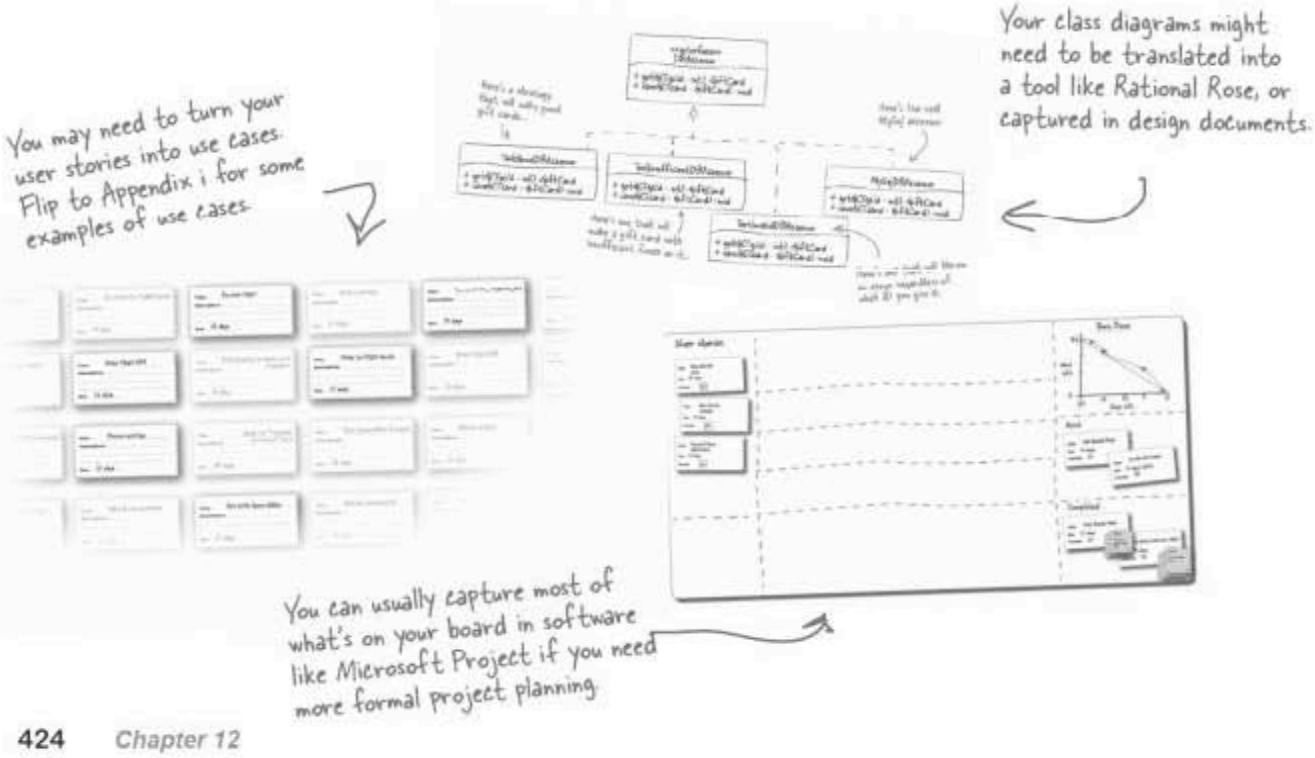
There are projects where you may need more formality than index cards and sticky notes. Some customers and companies want documents that are a little more formal. It's OK, though; everything you've learned still applies, and you don't need to scrap a process that's working just to dress up your development a bit.

First, remember that unless you absolutely have to, wait until the end of your current iteration to make any changes to your process. Next, know why you're making a change and how you're going to measure its effectiveness. "The customer won't pay me unless we have design documentation" is a perfectly reasonable starting point for dressing up your process. However, it's still important to know how you're going to measure effectiveness. Most customers are (rightfully) concerned about their business and aren't just looking to give you extra work.

If you're going to put together more documentation, project plans, use cases, or anything else, make sure it helps your customer—and hopefully your team—be better at communication. That's a result that is good for your project.

Do what you're doing...just prettier

Most of the work you're doing can be captured and reported in a more formal fashion. With software and a little extra polish, everything from your big board to your user stories can be converted into something that meets your customer's needs.



there are no Dumb Questions

Q: Isn't less formality better? Can't I convince my customer that index cards are all I need?

A: It's not about more formal versus less formal. It's about what works to get the right software written. The board with stories and tasks works well for lots of teams because it's simple, visual, and effective at communicating what needs to be done. It's not effective at lining up external teams that might be relying on your software or for when marketing should schedule the major release events and start shipping leaflets. Don't add formality for the sake of being formal, but there are times when you will need more than index cards.

Q: If we have to use a project planning tool, should I keep the board too?

A: Yes. There'll be some duplication of effort, but the board works so well with small teams that it's very hard to get anything more effective. The tangible tasks hanging on the board that team members physically move around just keeps the team in sync better than a screenshot or printout does.

Q: My customer wants design documentation and just doesn't get that my design just "evolves"...

A: Be careful with this one. Refactoring and evolutionary design work well with experienced teams who know their product, but it's very easy to get something wrong. On top of that, not giving your customer the design documentation they want is asking them to take a huge leap of faith in what you're doing—and that leap might not be justified yet. Most successful teams do at least some up-front design each iteration. You need to make sure the design documentation they're asking for is providing value, but design material is usually pretty useful for both you and the customer. Just make sure you account for the work in your estimates. Don't let TDD or "evolutionary design" be an excuse for "random code that I typed in late last night."

Q: My customer wants a requirements document, but user stories are working really well for my team. Now what?

A: If your customer has a history with more formal requirements documents, it may be very difficult to make the shift to user stories. In general, you don't want more than one requirement document directing how things should be implemented. It's very difficult to keep a document and user stories in sync, and someone always gets stuck resolving the conflicts.

Instead, try starting with a user story and at the end of the iteration break up the user story into "the user shall" statements that can fit into a formal requirements document. Or, if the customer wants nothing to do with user stories, you can try going the other direction: pull several "The user shall" type statements into a user story and work from the stories. But watch out—those "the user shall" type requirements often don't give you a lot of context about the application as a whole, and what it's doing.

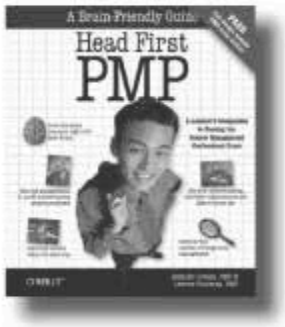
Neither approach is ideal, but one may be a compromise that's workable. You need to be absolutely diligent about changes in both directions, though.

Choose a process that works for YOUR team and YOUR project...

...and then tailor the artifacts it produces to match what YOUR customer wants and needs.

Some additional resources...

Even with all of the new tools available to you, there's always more to learn. Here are some places to go for some more great information on software development, and the techniques and approaches you've been learning about.



Head First PMP

If you've managing your team, there's more to good software—and project management—than just the big board. PMP takes you beyond the basics into a tried-and-true project management process—and help you get certified along the way.

Even if you've never considered yourself a project manager, if you're leading or in charge of a team, this book could help.

Test-driven development Yahoo! group

One of the all-time great resources for information on test-driven development is on the "Test-driven Development" group at Yahoo!. The group is pretty active, with current discussions and debates as well as some great historical information. You can find the group online at `http://tech.groups.yahoo.com/group/testdrivendevelopment/`.

Head First Object-Oriented Analysis and Design

Want to get deeper into code? To learn more about object-oriented principles of design and implementation? If you loved drawing class diagrams and implementing the strategy pattern, check out this book for a lot more on getting down deep with code.

Rational Unified Process web site

One of the founding iterative processes is the Rational Unified Process (RUP). It's a pretty heavy process out-of-the-box, but it's designed to be tailored to your needs. It's also a common approach to large-scale enterprise development. Be sure and read this and some Agile- or XP-leaning sites, so you get a balanced picture. Check it out online at `http://www-306.ibm.com/software/awdtools/rup/`.

The Agile Alliance

The Agile Alliance is a great kickoff point for information on Agile processes like extreme programming, Scrum, or Crystal. Agile processes are very lightweight, and you'll see many of the things you learned about, albeit from a different perspective at times. Check it out at `http://www.agilealliance.org/`.

More knowledge == better process

There are tons more resources than just these. Part of good software development is keeping on top of what's going on. And that means reading, Googling, asking your buddies on other projects—anything you can do to find out what other people are doing, and what works for them.

And never be afraid to try something new, even for just an iteration. You never know what might work, or what you might pick up that's just perfect for **your** project.

Tools for your Software Development Toolbox

Software Development is all about developing and delivering great software. In this chapter, you got some additional resources to help you take your knowledge out into the real world. For a complete list of tools in the book, see Appendix ii.

Development Techniques

Critically evaluate any changes to your process with real metrics.

Formalize your deliverables if you need to, but always know how it's providing value.

Try hard to only change your process between iterations.

Here are some of the key techniques you learned in this chapter...

...and some of the principles behind those techniques.

Development Principles

Good developers develop—great developers ship.

Good developers can usually overcome a bad process.

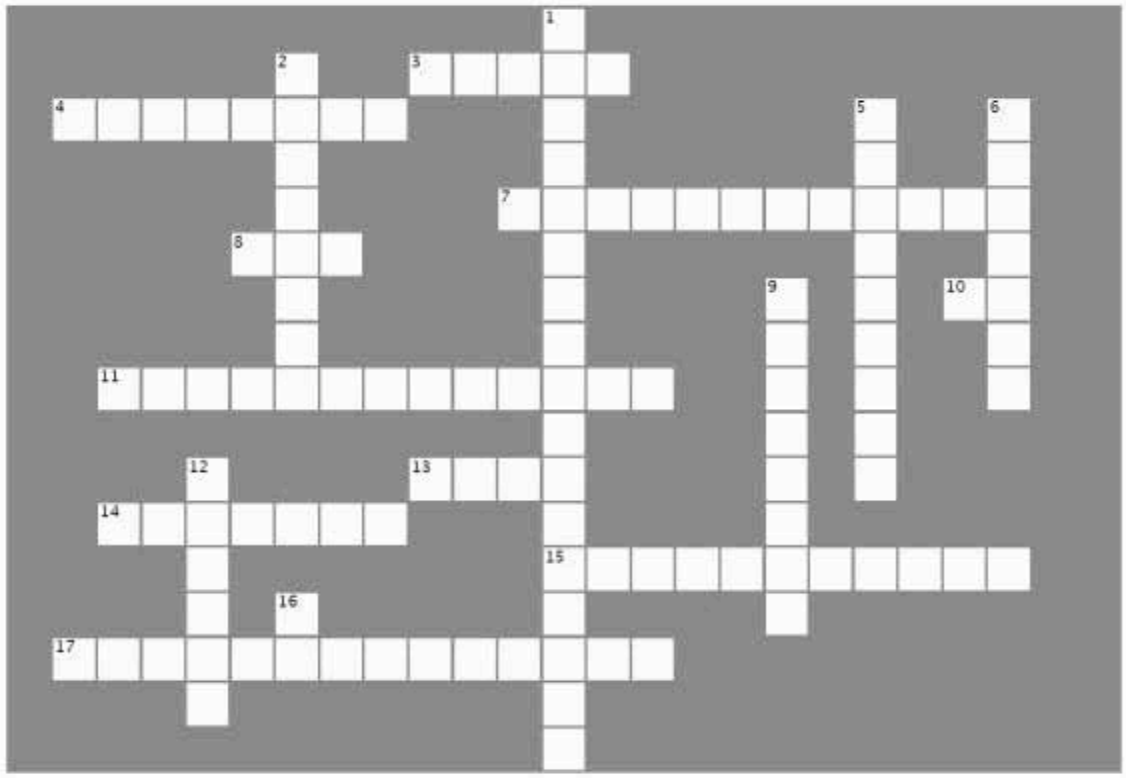
A good process is one that lets your team be successful.

BULLET POINTS

- **Take your team's opinion into account** whenever you're going to make changes to the process; they have to live with your changes, too.
- Any process change should show up **twice**: once to **decide to do it** and once to **evaluate whether or not it worked**.
- Steer clear of **more than one place to store requirements**. It's always a maintenance nightmare.
- Be **skeptical** of magic, **out-of-the-box processes**. Each project has something **unique** to it, and your process should be flexible.

SoftwareDevelopmentcross

This is it, the last crossword. This time the solutions are from anywhere in the book.



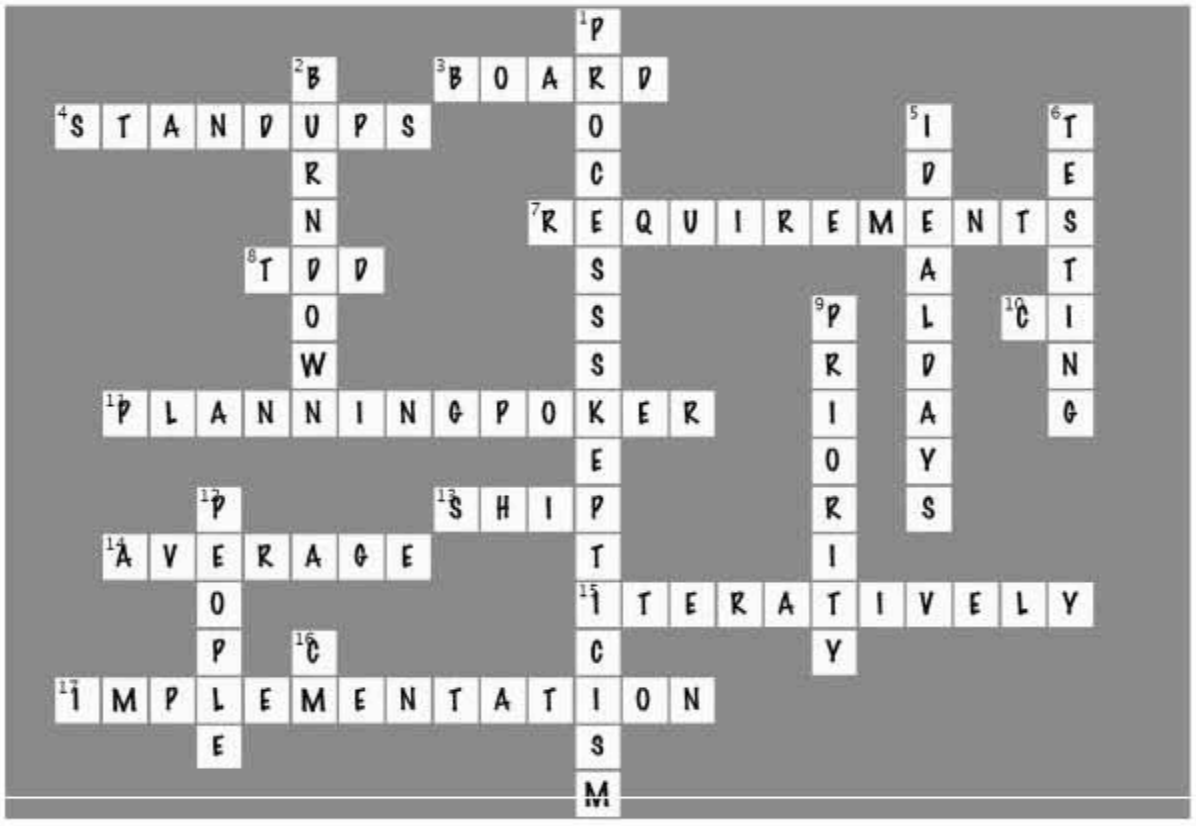
Across

3. Project planning tools can help with projections and presentation of schedule, but do them in parallel with your .....
4. No more than 15 minutes, these keep the team functioning as a team toward a common goal.
7. Every iteration involves .....
8. This is an approach where you write your tests first and refactor like mad.
10. This is a process that checks out your code, builds it, and probably runs tests.
11. High stakes game of estimation.
13. Good Developers develop, Great developers .....
14. The team member you should estimate for.
15. No matter what process you pick, develop .....
17. Every iteration involves .....

Down

1. This means to evaluate processes critically and demand results from each of the practices they promote.
2. Shows how you're progressing through an iteration.
5. What you should be estimating in.
6. Every iteration involves .....
9. How you rack and stack your user stories.
12. The greatest indicator of success or failure on a project.
16. This is a process that tracks changes to your code and distributes them among developers.

SoftwareDevelopmentcross



It's time to leave a mark on the ~~board~~ world!

There are exciting times ahead! Armed with all of your software development knowledge, it's time to put what you know to work... so get out there and change the world. Don't forget that the realm of software never stops changing, either. Keep reading, learning, and please, if you can schedule it in your iteration, swing by Head First Labs (*www.headfirstlabs.com*) and drop us a note on how these tools have helped you out.

And be sure and move your "Visit Head First Labs" task to Completed when you're through.

The top 5 topics
(we didn't cover)

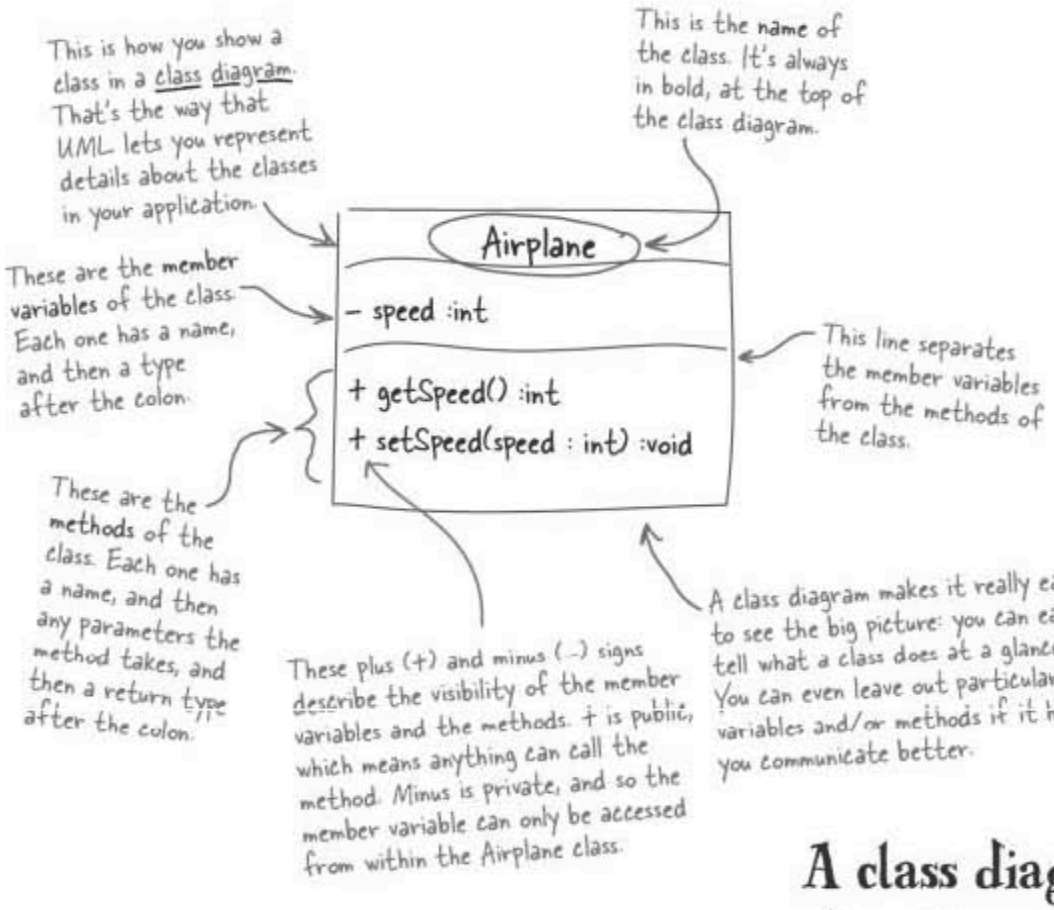
Ever feel like something's missing? We know what you mean...

Just when you thought you were done... there's more. We couldn't leave you without a few extra things, things we just couldn't fit into the rest of the book. At least, not if you want to be able to carry this book around without a metallic case and castor wheels on the bottom. So take a peek and see what you (still) might be missing out on.

#1. UML class diagrams

When you were developing the iSwoon application in Chapters 4 and 5, we described the design using UML, the ***Unified Modeling Language***, which is a language used to communicate just the **important details** about your **code** and **application's structure** that other developers and customers need, without getting into things that *aren't* necessary.

UML is a great way of working through your design for iSwoon without getting too bogged down in code. After all, it's pretty hard to look at 200 lines of code and focus on the big picture.



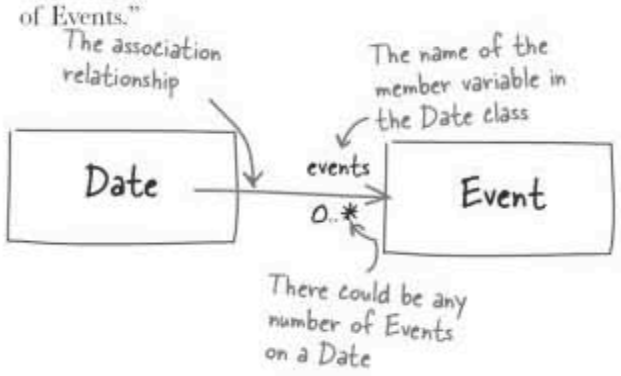
A class diagram describes the static structure of your classes.

Class diagrams show relationships

Classes in your software don't exist in a vacuum, they interact with each other at runtime and have relationships to each other. In this book you've seen two relationships, called association and inheritance.

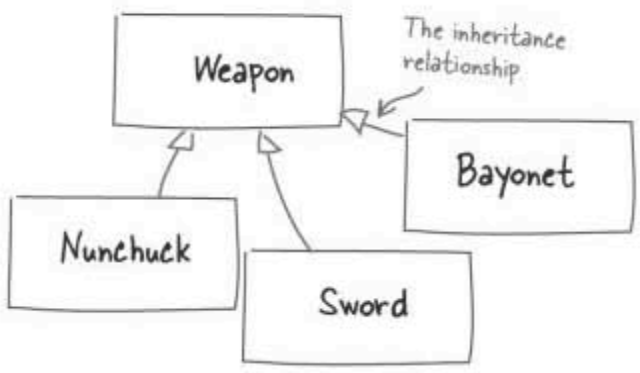
Association

Association is where one class is made up of objects of another class. For example, you might say "A Date is associated with a collection of Events."



Inheritance

Inheritance is useful when a class inherits from another class. For example, you might "A Sword inherits from Weapon."



there are no Dumb Questions

Q: Don't I need a big expensive set of tools to create UML diagrams?

A: No, not al all. The UML language was originally designed such that you could jot down a reasonably complex design with just a pencil and some paper. So if you've got access to a heavyweight UML modeling tool then that's great, but you don't actually need it to use UML.

Q: So the class diagram isn't a very complete representation of a class, is it?

A: No, but it's not meant to be. Class diagrams are just a way to communicate the basic details of a class's variables and methods. It also makes it easy to talk about code without forcing you to wade through hundreds of lines of Java, or C, or Perl.

Q: I've got my own way of drawing classes; what's wrong with that?

A: There's nothing wrong with your own notation, but it can make things harder for other people to understand. By using a standard like UML, we can all speak the same language and be sure we're talking about the same thing in our diagrams

Q: So who came up with this UML deal, anyway?

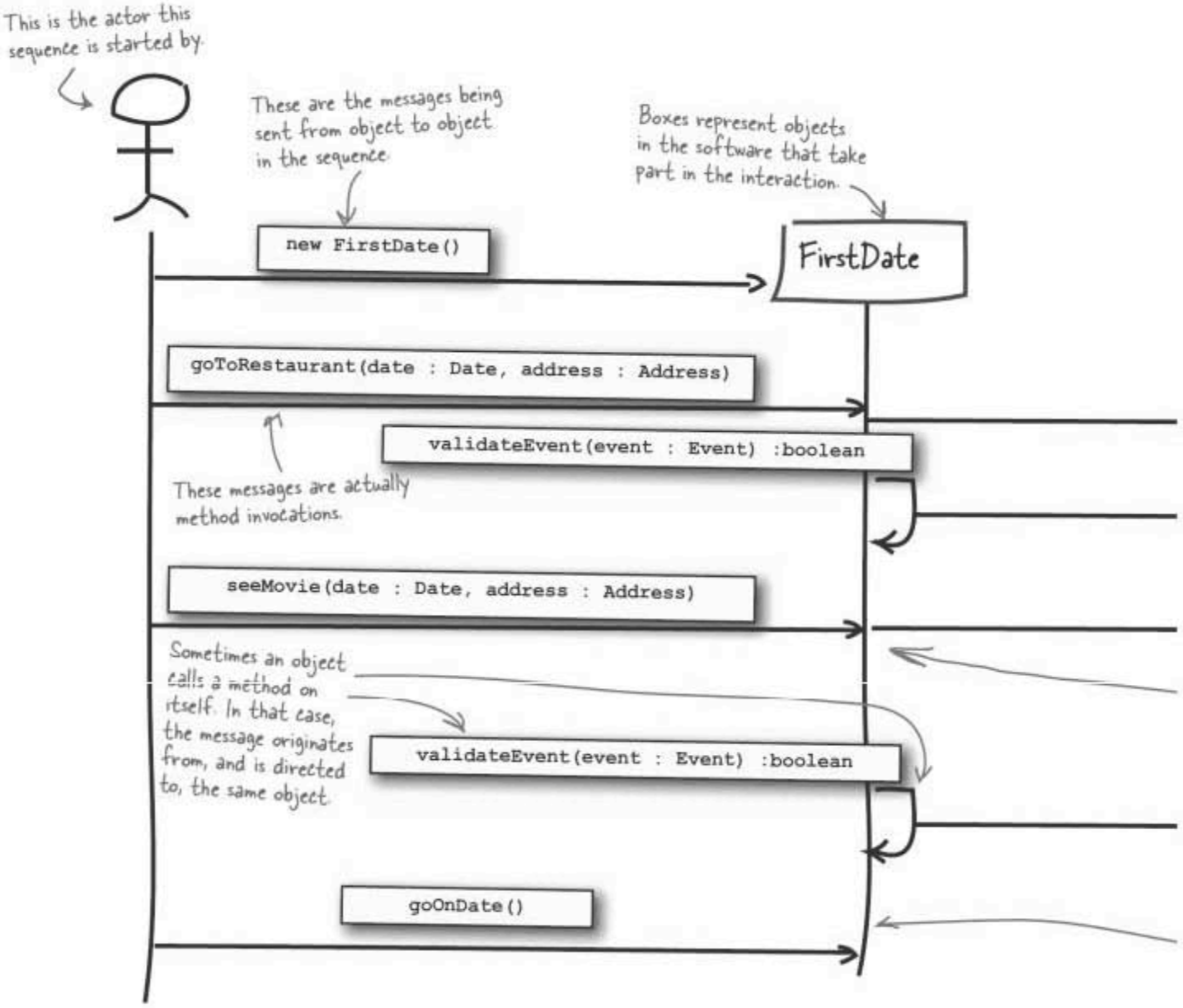
A: The UML specification was developed by Rational Software, under the leadership of Grady Booch, Ivar Jacobson, and Jim Rumbaugh (three really smart guys). These days it's managed by the OMG, the Object Management Group.

Q: Sounds like a lot of fuss over that simple little class diagram thing.

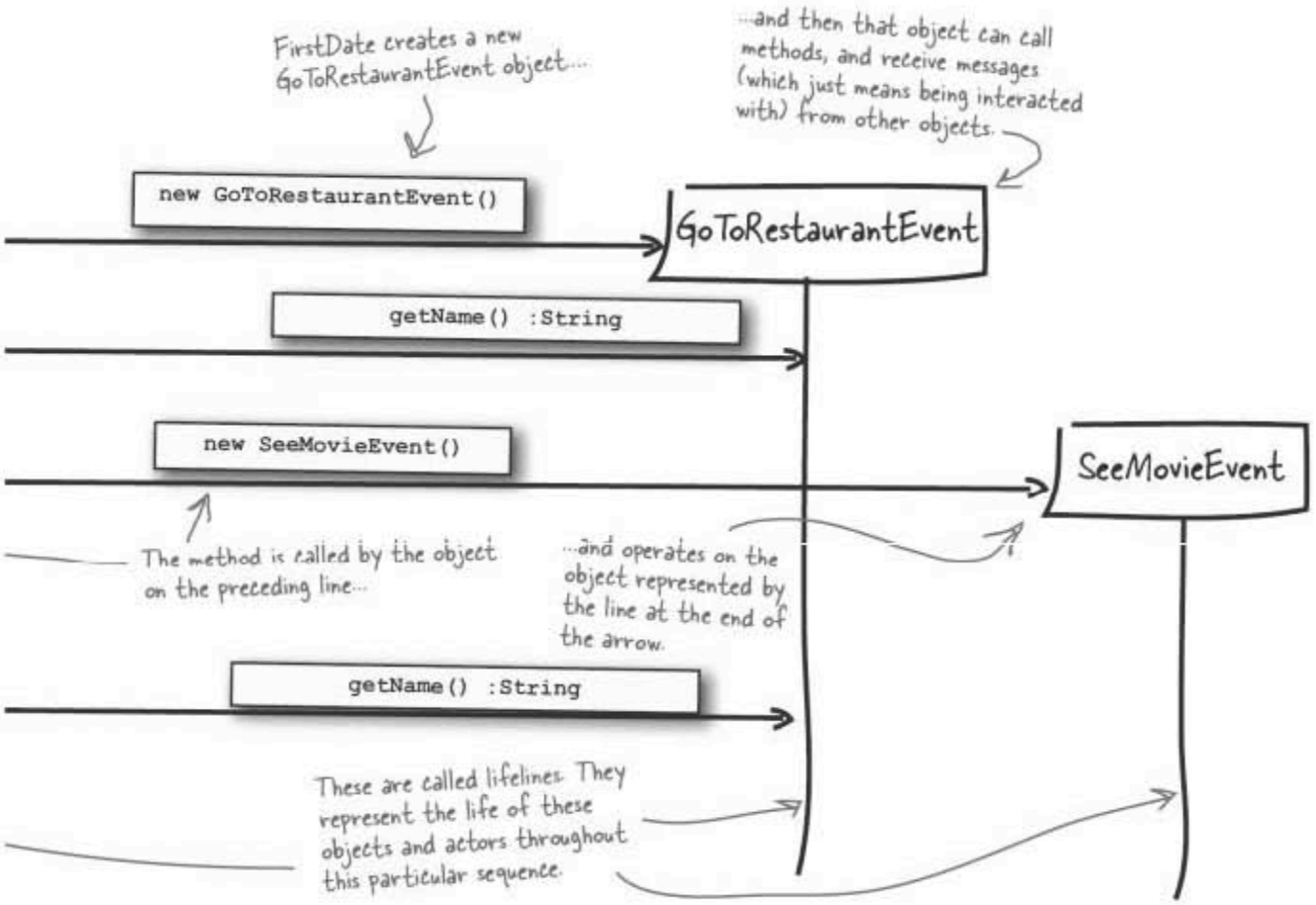
A: UML is actually a lot more than that class diagram. UML has diagrams for the state of your objects, the sequence of events in your application, and it even has a way to represent customer requirements and interactions with your system. And there's a lot more to learn about class diagrams, too.

#2. Sequence diagrams

A static class diagram only goes so far. It shows you the classes that make up your software, but it doesn't show how those classes work together. For that, you need a UML **sequence diagram**. A sequence diagram is just what it sounds like: a visual way to show the order of events that happen, such as invoking methods on classes, between the different parts of your software.



Sequence diagrams show how your objects interact at runtime to bring your software's functionality to life.



#3. User stories and use cases

You used user stories throughout this book to capture your requirements. User stories are really great at getting a neat description of exactly what the customer needs your software to do. But a lot of more formal processes recommend something called a **use case**.

Luckily, there's easily enough overlap between user stories and use cases for you to use either technique to capture your customer's requirements:

A user story that describes sending a picture using the Beatbox software from chapter 6.

Title: Send a picture to other users

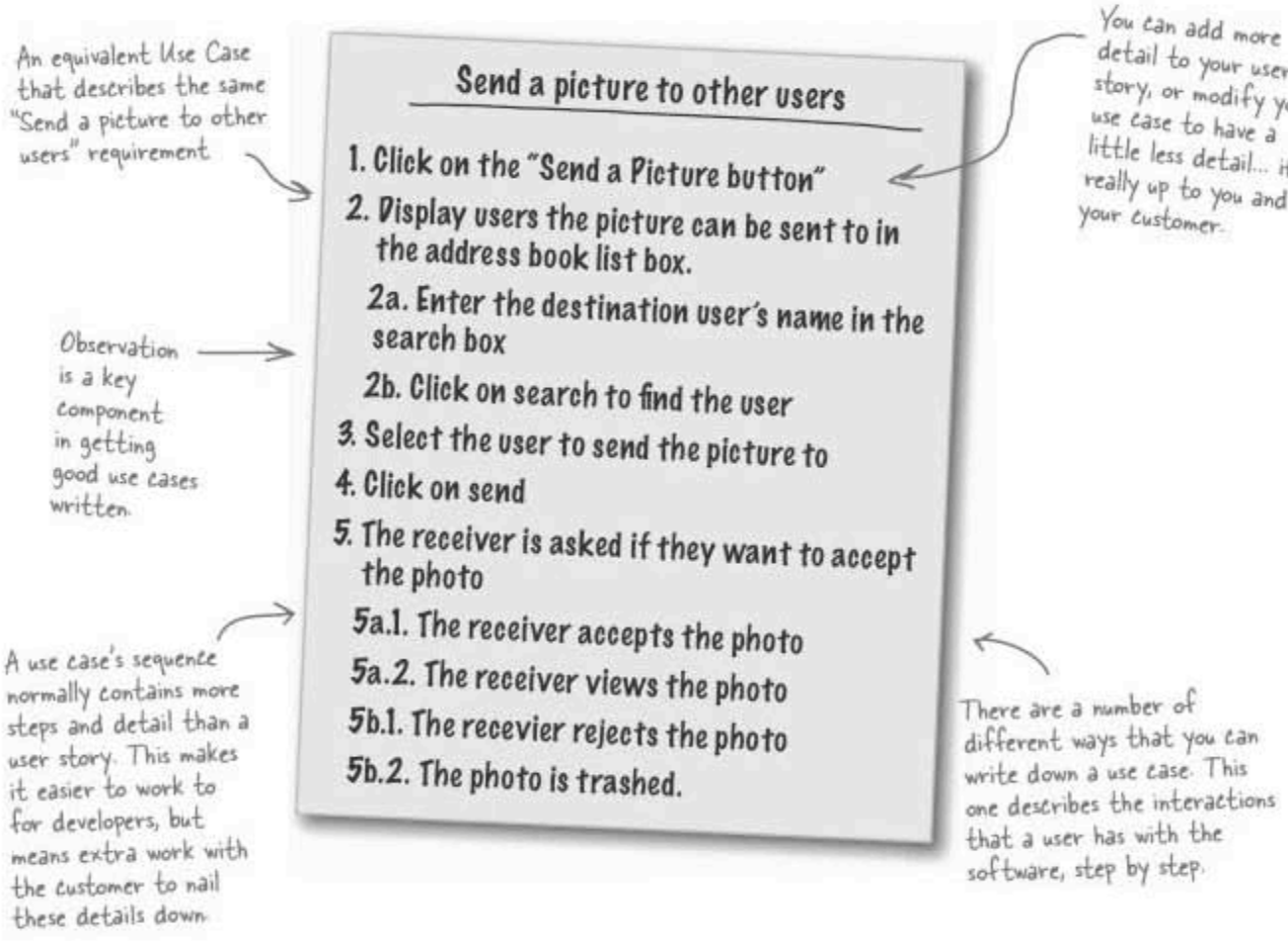
Description: Click on the "Send a Picture" button to send a picture (only JPEG needs to be supported) to the other users. The other user should have the option to not accept the file. There are no size limits on the file being sent.

Priority: 20 **Estimate:** 4

A user story that has been prepared for your project board contains a priority and an estimate

A user story and a use case describe ONE THING that your software needs to do.

...and only one thing



So what's the big difference?

Well, actually not a lot, really. User stories are usually around three lines long, and are accompanied by an estimate and a priority, so the information is all in one bite-sized place. Use cases are usually reasonably more detailed descriptions of a user's interaction with the software. Use cases also aren't usually written along with a priority or an estimate—those details are often captured elsewhere, in more detailed design documentation.

User stories are ideally written by the customer, whereas traditionally use cases are not. Ultimately either approach does the same job, capturing what your customer needs your software to do. And one use case, with alternate paths (different ways to use the software in a specific situation) may capture more than one user story.

#4. System tests vs. unit tests

In chapters 7 and 8, you learned how to build testing and continuous integration into your development process. Testing is one of the key tools you have to prove that **your code works** and **meets the requirements** set by your customer. These two different goals are supported by two different types of tests.

Unit testing is at a very low level... source files and XML descriptors.

Unit tests test your CODE

Unit tests are used to test that your code ***does what it should***. These are the tests that you build right into your continuous build and integration cycle, to make sure that any changes that you make to code don't break these tests, on your code and the rest of the code base.

Ideally, every class in your software should have an associated unit test. In fact, with test-driven development, your tests are developed before any code is even written, so there is no code without a test. Unit tests have their limits, though. For example, maybe you make sure that calling `drive()` on the `Automobile` class works... but what happens when other instances of `Automobile` are also driving, and using the same `RaceTrack` object, too?

System tests test your SOFTWARE

System tests pick up where unit tests leave off. A system test tests your code when it is integrated into a **fully functional system**. System tests are sometimes automated, but often involve someone actually exercising your entire system in very much the same way as the end user will.

For example, you might fire up the GUI for monitoring a race, press the "Start Race" button, watch animated versions of cars spin around the track, and then initiate a wreck. Does everything work the way the customer expects? That's a system test.

System testing looks at your application as a whole.

there are no Dumb Questions

Q: In addition to unit and system tests, aren't there lots of other types of tests as well?

A: Yes. Testing is a BIG field of work. There are various names for testing, conducted at anything from the source code level to enterprise software integration level. For example, you may hear of **acceptance tests**. Acceptance tests are often conducted with the customer, where the customer either accepts or rejects your software as doing what they need..

Unit tests prove that your code WORKS. System tests prove that your software meets its REQUIREMENTS.

#5. Refactoring

Refactoring is the process of modifying the structure of your code, ***without*** modifying its behavior. Refactoring is done to increase the cleanness, flexibility, and extensibility of your code, and usually is related to a **specific improvement in your design**.

Most refactorings are fairly simple, and focus on one specific design aspect of your code. For example:

```
public double getDisabilityAmount() {
  // Check for eligibility
  if (seniority < 2)
    return 0;
  if (monthsDisabled > 12)
    return 0;
  if (isPartTime)
    return 0;
  // Calculate disability amount and return it
}
```

While there's nothing particularly wrong with this code, it's not as maintainable as it could be. The **`getDisabilityAmount()`** method is really doing two things: checking the eligibility for disability, and then calculating the amount.

By now, you should know that violates the Single Responsibility Principle. We really should separate the code that handles eligibility requirements from the code that does disability calculations. So we can *refactor* this code to look more like this:

```
public double getDisabilityAmount() {
  // Check for eligibility
  if (isEligibleForDisability()) {
    // Calculate disability amount and return it
  } else {
    return 0;
  }
}
```

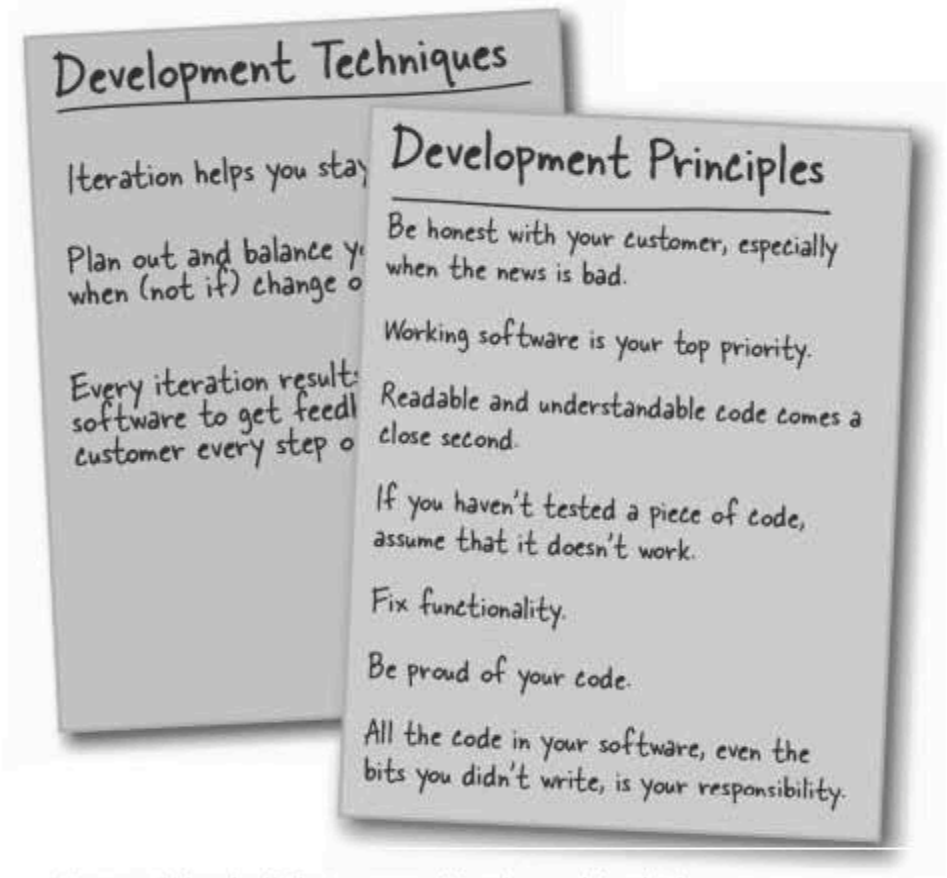
We've taken two responsibilities, and placed them in two separate methods, adhering to the SRP.

Now, if the eligibility requirements for disability change, only the **`isEligibleForDisability()`** methods needs to change—and the method responsible for calculating the disability amount doesn't.

Think of refactoring as a checkup for your code. It should be an ongoing process, as code that is left alone tends to become harder and harder to reuse. Go back to old code, and refactor it to take advantage of new design techniques you've learned. The programmers who have to maintain and reuse your code will thank you for it.

Refactoring changes the internal structure of your code WITHOUT affecting your code's behavior.

Tools for the experienced software developer



Ever wished all those great tools and techniques were in one place? This is a roundup of all the software development **techniques** and **principles** we've covered. Take a look over them all, and see if you can **remember what each one means**. You might even want to **cut these pages out** and tape them to the bottom of your **big board**, for everyone to see in your daily standup meetings.

Development Techniques

CHAPTER 1

Iteration helps you stay on course

Plan out and balance your iterations when (not if) change occurs

Every iteration results in working software and gathers feedback from your customer every step of the way

CHAPTER 2

Bluesky, Observation, and Roleplay to figure out how your system should behave

Use user stories to keep the focus on functionality

Play planning poker for estimation

CHAPTER 3

Iterations should ideally be no longer than a month. That means you have 20 working calendar days per iteration

Applying velocity to your plan lets you feel more confident in your ability to keep your development promises to your customer

Use (literally) a big board on your wall to plan and monitor your current iteration's work

Get your customer's buy-in when choosing what user stories can be completed for Milestone 1.0, and when choosing what iteration a user story will be built in

CHAPTER 4

You didn't think the exercises were over, did you? Write your own techniques for Chapters 4 and 5.

CHAPTER 5

CHAPTER 6

Use a version control tool to track and distribute changes in your software to your team

Use tags to keep track of major milestones in your project (ends of iterations, releases, bug fixes, etc.)

Use branches to maintain a separate copy of your code, but only branch if absolutely necessary

CHAPTER 6.5

Use a build tool to script building, packaging, testing, and deploying your system

Most IDEs are already using a build tool underneath. Get familiar with that tool, and you can build on what the IDE already does

Treat your build script like code and check it into version control

CHAPTER 7

There are different views of your system, and you need to test them all

Testing has to account for success cases as well as failure cases

Automate testing whenever possible

Use a continuous integration tool to automate building and testing your code on each commit

CHAPTER 8

Write tests first, then code to make those tests pass

Your tests should fail initially; then after they pass you can refactor

Use mock objects to provide variations on objects that you need for testing

CHAPTER 9

Pay attention to your burn-down rate—especially after the iteration ends

Iteration pacing is important—drop stories if you need to keep it going

Don't punish people for getting done early—if their stuff works, let them use the extra time to get ahead or learn something new

CHAPTER 10

What did you learn in Chapter 10? Write it down here.

CHAPTER 11

Before you change a single line of code, make sure it is controlled and buildable

When bugs hit code you don't know, use a spike test to estimate how long it will take to fix them

Factor in your team's confidence when estimating the work remaining to fix bugs

Use tests to tell you when a bug is fixed

CHAPTER 12

Critically evaluate any changes to your process with real metrics

Formalize your deliverables if you need to, but always know how it's providing value

Try hard to only change your process between iterations

Development Principles

CHAPTER 1

Deliver software that's needed

Deliver software on time

Deliver software on budget

CHAPTER 2

The customer knows what they want, but sometimes you need to help them nail it down

Keep requirements customer-oriented

Develop and refine your requirements iteratively with the customer

CHAPTER 3

Keep iterations short and manageable

Ultimately, the customer decides what is in and what is out for Milestone 1.0

Promise, and deliver

ALWAYS be honest with the customer

CHAPTER 4

CHAPTER 5

Be the author... write your own principles based on what you learned in Chapter 5.

CHAPTER 6

Always know where changes should (and shouldn't) go

Know what code went into a given release—and be able to get to it again

Control code change and distribution

CHAPTER 6.5

Building a project should be repeatable and automated

Build scripts set the stage for other automation tools

Build scripts go beyond just step-by-step automation and can capture compilation and deployment logic decisions

We didn't add any techniques and principles to Chapter 4... can you come up with a few and write them here?

CHAPTER 7

Testing is a tool to let you know where your project is at all times

Continuous integration gives you confidence that the code in your repository is correct and builds properly

Code coverage is a much better metric of testing effectiveness than test count

CHAPTER 8

TDD forces you to focus on functionality

Automated tests make refactoring safer; you'll know immediately if you've broken something

Good code coverage is much more achievable in a TDD approach

CHAPTER 9

Iterations are a way to impose intermediate deadlines—stick to them

Always estimate for the ideal day for the average team member

Keep the big picture in mind when planning iterations—and that might include external testing of the system

Improve your process iteratively through iteration reviews

CHAPTER 10

Chapter 10 was all about third-party code. What principles did you pick up?

CHAPTER 11

Be honest with your customer, especially when the news is bad

Working software is your top priority

Readable and understandable code comes a close second

If you haven't tested a piece of code, assume that it doesn't work

Fix functionality

Be proud of your code

All the code in your software, even the bits you didn't write, is your responsibility

CHAPTER 12

Good developers develop—great developers ship

Good developers can usually overcome a bad process

A good process is one that lets YOUR team be successful